MCS®-51 UTILITIES USER'S GUIDE FOR DOS SYSTEMS

Order Number: 122748-001

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	3/86	M.K.S.

This manual describes the RL51 linker and locator and the LIB51 librarian for program modules produced by MCS-51 language translators such as ASM51 and PL/M-51.

The RL51 and the LIB51 program operate on an Intel development system with an 8080 or 8085 processor. The configuration must include 64K of RAM, a console, and at least one diskette or hard disk drive.

NOTE

In this manual, the term MCS-51 refers to all members of the MCS-51 family of microcomputers and to the software development tools for the MCS-51 family.

Reader's Guide

The manual is organized into six chapters and five appendixes:

Chapter 1 discusses the advantages of modular programming and summarizes the process of modular programming with the MCS-51 development tools.

Chapter 2 reviews the mechanics of linkage and location for the RL51 program.

Chapter 3 gives the details on invoking the linker/locator.

Chapter 4 discusses the files and displays produced by the RL51 program, with examples.

Chapter 5 contains three examples of programs, with the link and locate steps for each program.

Chapter 6 describes the LIB51, the MCS-51 library manager and its usage.

Appendix A presents the syntax of the RL51 commands with brief definitions of the controls.

Appendix B lists the error messages and warnings displayed by RL51, with suggestions for corrective action.

Appendix C lists a summary of LIB51 commands.

Appendix D lists the error messages generated by LIB51, with suggestions for corrective action.

Appendix E contains hexadecimal-decimal conversion tables as a convenient reference.

Related Publications

The following list provides the manual title and order number for all Intel software development tools that run on DOS systems. Note that some manuals have two formats and two order numbers. One version of the manual is provided in a binder. This version is not sold separately; it can only be purchased when purchasing a software product. The second version, which has a soft cover, is sold separately. Use the soft cover number when ordering a manual separately.

Manual Title	Binder	Soft Cover
MCS®-51 Macro Assembler User's Guide for DOS Systems	122753	122752
PL/M-51 User's Guide for DOS Systems	122742	122743
MCS®-51 Family of Single-Chip Microcomputers User's Manual for DOS Systems		121517
MCS®-51 Macro Assembly Language Pocket Reference for DOS Systems		122755
MCS®-51 Utilities Pocket Reference for DOS Systems		122750

Notational Conventions

UPPERCASE Characters shown in uppercase must be entered in the order shown. Enter the command words as shown, or use a system-defined abbreviation. You may enter the characters in uppercase or lowercase.

italic Italic indicates a meta symbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:

directory-name Is that portion of a *pathname* that acts as a file locator by identifying the device and/or directory containing the *filename*.

filename Is a valid name for the part of a *pathname* that names a file.

system-id Is a generic label placed on sample listings where an operating system-dependent name would actually be printed.

Vx.y Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.

[ ] Brackets indicate optional arguments or parameters.

{ } One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.

{ }... At least one of the enclosed items must be selected unless the field is also surrounded by brackets, in which case it is optional. The items may be used in any order unless otherwise noted.

| The vertical bar separates options within brackets [ ] or braces { }.

... Ellipses indicate that the preceding argument or parameter may be repeated.

[...] The preceding item may be repeated, but each repetition must be separated by a comma.

punctuation Punctuation other than ellipses, braces, and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered:

SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')

input lines In interactive examples, user input lines are printed in white on black to differentiate them from system output.

<cr> Indicates a carriage return.

Appendix A
Summary of RL51 Controls

Appendix B
RL51 Error Messages

Appendix C
LIB51 Command Summary

Appendix D
LIB51 Error Messages

Appendix E
Hexadecimal-Decimal Conversion Table

Index

Figures

Tables

The Advantages of Modular Programming

Many programs are too long or complex to write as a single unit. Programming becomes much simpler when the code is divided into small functional units. Modular programs are usually easier to code, debug, and change than monolithic programs.

The modular approach to programming is similar to the design of hardware that contains numerous circuits. The device or program is logically divided into "black boxes" with specific inputs and outputs. Once the interfaces between the units have been defined, detailed design of each unit can proceed separately.

Efficient Program Development

Programs can be developed more quickly with the modular approach because small subprograms are easier to understand, design, and test than large programs. With the module inputs and outputs defined, the programmer can supply the needed input and verify the correctness of the module by examining the output. The separate modules are then linked and located into one program module. Finally, the completed program is tested.

Multiple Use of Subprograms

Code written for one program is often useful in others. Modular programming allows these sections to be saved for future use. Because the code is relocatable, saved modules can be linked to any program that fulfills their input and output requirements. With monolithic programming, such sections of code are buried inside the program and are not so available for use by other programs.

If you put your frequently-used subprograms into a library, RL51 will take care to load only those you need. Thus, you can save RAM and ROM without having to keep track of what is needed and what is not.

Ease of Debugging and Modifying

Modular programs are generally easier to debug than monolithic programs. Because the modular interfaces are well-defined, problems can be isolated to specific modules. Once the faulty module has been identified, fixing the problem is considerably simpler. When a program must be modified, modular programming simplifies the job. You can link new or modified modules to the existing program with confidence that the rest of the program will not be changed.

MCS®-51 Modular Program Development Process

This section is a brief review of the program development process using an MCS-51 language translator (e.g., the relocatable MCS-51 assembler or PL/M-51 compiler), linker/locator, code converter programs, PROM programmer, and ICE™-51 in-circuit emulator. The process is shown in figure 1-1.
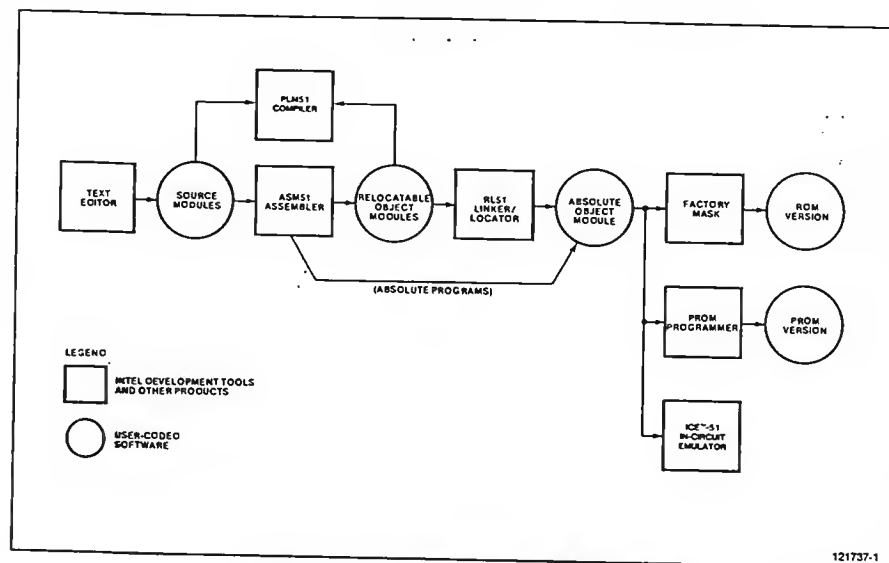
Figure 1-1. MCS®-51 Program Development Process

Segments, Modules, Libraries, and Programs

In the initial design stages, the tasks to be performed by the program are defined and then partitioned into subprograms. Here are brief introductions to the kinds of subprograms used with the MCS-51 assembler and linker/locator.

A segment is a unit of code or data memory. A segment may be relocatable or absolute. A relocatable segment in a module can be a complete segment or can be a "partial" segment to be combined with other partial segments from other modules. A relocatable segment has a name, type, and other attributes that allow the linker to combine it with other partial segments, if required, and to correctly locate the segment. An absolute segment has no name and cannot be combined with other segments. See Chapter 2 for more detail on partial segments.

A module contains one or more segments or partial segments. A module has a name assigned by the user. The module definitions determine the scope of local symbols. An object file contains one or more modules. You can add modules to a file by transfering the new modules from their individual files to another file.

A library is a file that contains one or more modules. A library file is internally marked as a library, so RL51 can easily identify it as such. RL51 selects, out of the modules in the library, only those previously referenced. Libraries are created using the LIB51 utility, which is described in detail in Chapter 6.

A program consists of a single absolute module, merging all absolute and relocatable segments from all input modules. The name of the output module produced by RL51 can be defined by the user or allowed to default to the name of the first input module.

Entering and Editing Source Modules

After the design is completed, use the text editor on your system to code the modules into source files. The source modules are coded in assembly language or a high-level language such as PL/M-51. The editor may also be used to make corrections in the source code.

Assembly and Compilation

The assembler (ASM51) and compiler (PL/M-51) translate the source code into relocatable object code, producing an object file. The ASM51 object file is relocatable when at least one input segment is relocatable; otherwise the object file is an absolute file. The PL/M-51 object file is always relocatable. The assembler and compiler also produce a listing file showing the results of the translation. When the ASM51 or PL/M-51 invocation contains the DEBUG control, the object file also receives the symbol table and other debug information for use in symbolic debugging of the program.

Relocation and Linkage

After translation of all modules of the program, the linker/locator, RL51, processes the object module files. The RL51 program combines relocatable partial segments with the same name, then assigns absolute memory locations to all the relocatable segments. RL51 also resolves all references between modules, using the library files when they are necessary for this resolution. RL51 outputs an absolute object module file that contains the completed program, and a summary listing file showing the results of the link/locate process, including a memory map, symbol table, and, optionally, an inter-module cross-reference (IXREF) listing.

ROM and PROM Versions

The absolute object module produced by RL51 can be loaded into members of the MCS-51 family of microcomputers. For ROM versions of the microcomputer, the program is masked into ROM during the manufacturing process. For PROM versions and versions with no on-chip CODE memory, a PROM programmer is used to load the absolute module into program memory accessible to the microcomputer for execution. Refer to the *MCS-51 Family of Single Chip Microcomputers User's Manual* for details on the versions of microcomputers available.

ICE™-51 In-Circuit Emulator

The ICE-51 in-circuit emulator is used for software and hardware debugging and integration into the final product. The absolute object modules produced by RL51 can be loaded into the emulator's memory for execution. Refer to the ICE-51 manual listed in the preface for details.

Keeping Track of Files

It is convenient to use the extensions of filenames to indicate the stage in the process represented by the contents of each file. Thus, source code files can use extensions like .SRC, .A51, or .P51 (indicating that the code is for input to ASM51 or PL/M-51). Object code files receive the extension .OBJ by default or the user can specify another extension. Executable files generally have no extension. Listing files

can use .LST, the default extension given by the translator. RL51 uses .M51 for the default listing file extension (in order not to destroy the ASM51 listing file with the .LST extension).

Library files customarily have the extension .LIB.

Use caution with the extension .TMP, as many utilities (including RL51 and LIB51) create temporary files with this extension. These utilities will delete your file if it has the same name and extension as the temporary files they create.

The Mechanics of Linkage and Location with RL51

2

This chapter describes the operation of the RL51 program. Most of the process is transparent to the user; however, an understanding of the operation at the level presented here will help you to use the linking and locating controls in the RL51 invocation. More specific details on the allocating process appear in Chapter 3.

Major Functions

The RL51 program performs the following major functions:

1. Selects modules (including library processing)
2. Combines relocatable partial segments of the same name into a single segment
3. Allocates memory for the combined segments resulting from the previous step, and for all other complete relocatable segments from the input modules
4. Overlays data segments
5. Resolves external symbol references between the input modules
6. Binds relocatable addresses to absolute addresses
7. Produces an absolute object file
8. Produces a listing file consisting of a link summary, a symbol table, and an IXREF report
9. Detects and lists errors found in the input modules or in the RL51 command invocation

Functions 1, 2, 3, 5, and 6 are described in the remainder of this chapter. Functions 7, 8, and 9 are discussed in Chapter 4; the RL51 command invocation and overlaying of data segments are described in Chapter 3.

Selecting Modules

Input files are processed in the order they are specified in the invocation command.

The processing of an input file varies according to the content—that is, whether it is a library or non-library file. A non-library file may contain a concatenation of zero or more object modules. A library file contains zero or more object modules together with control information. A module in a non-library file is processed if it was explicitly listed in the module list, or if the module list was not specified at all (in other words, as if all modules were listed implicitly).

The processing of a library file is more complicated. If a module list was specified for the library file, then it is processed in the same manner as a non-library file. If a module list was not specified, then the library file is processed only if the previously processed modules contained at least one unresolved external. The library is scanned for modules containing public symbols that match as yet unresolved externals. Each such module is processed as if it were explicitly specified. The selection process continues until the modules in the library cannot satisfy any unresolved externals (including any externals encountered while processing modules from the library).

RL51 will report an error if the same module name is encountered more than once during the link process.

Take TRIGON.LIB as an (utterly fictitious) example. Assume it contains procedures called SINE, COSINE, TANGENT, ARCSINE, ARCCOS, ARCTAN, HYPER-BOLIC_SINE, and HYPERBOLIC_COSINE.

When RL51 starts processing TRIGON.LIB, it has already made a first pass over all files that appear before it in the invocation line. If one of these contains a reference to the external SINE, and there is no public by that name, RL51 will assume that the procedure SINE from TRIGON.LIB is to be loaded. Otherwise, it will leave SINE alone for the moment.

If, while loading from TRIGON.LIB, RL51 encounters new externals that a module in the library can resolve, it will scan the library once more. Thus, there is no logical order among modules in a library; they are all equal. If TANGENT calls SINE and COSINE, and they are in the same library, in any order whatsoever, a reference to TANGENT will cause all three to be loaded.

Partial Segments

A segment is a unit of code memory or data memory. The portion of a segment defined in one module is called a partial segment. A partial segment has the following attributes (defined in the source module):

*   Name. A relocatable segment has a name by which it is linked with other portions of the same segment from other modules. Absolute segments do not have names.
*   Type. The type identifies the address space to which a segment belongs: CODE, XDATA, DATA, IDATA, or BIT.
*   Relocatability. For relocatable segments only, this attribute describes any special constraints on relocation (PAGE, INPAGE, BLOCK, BITADDRESSABLE, or UNIT).
*   Size. The size of the segment in bytes or bits, depending on the type.
*   Base Address. The lowest address in the partial segment. For absolute segments, the base address is assigned at assembly time; for relocatable segments, it is assigned at location time.

Absolute segments are complete segments; they are taken as is into the output module. Relocatable segments are either defined by ASM51 users (using the SEGMENT directive in the source module) or automatically generated by the PL/M-51 compiler.

Refer to the *MCS-51 Macro Assembler User's Guide* for details on the assembler directives.

Combining Relocatable Segments

After processing the invocation command, RL51 performs a first pass over the input modules identified in the command. Pass I generates a segment table, a publics table, and an unresolved externals table. The segment table is discussed in this section; the other two tables are discussed later in this chapter.

The segment table contains the name, length, type, and relocation attribute of all combined segments from all modules. Combined segments are produced from the partial segments in the input modules according to the following rules:

*   RL51 combines all partial segments with the same name into one relocatable segment. For example, if three input modules each have a partial relocatable segment named STACK, the segment table will have one segment named STACK that combines the length of the three partial segments.

*   All the partial segments to be combined must be of the same type (CODE, DATA, IDATA, XDATA, or BIT). If any partial segments have the same names but different types, an error occurs.
*   The length of the combined segment must not exceed the physical size of the memory type. Details on maximum size appear later in this chapter.
*   The relocation attributes of all the partial segments to be combined must either be the same or UNIT-aligned combined with one other attribute. The combined segment receives the relocation attribute shared by the input partial segments, or, if the segments have attribute UNIT-aligned combined with one other attribute, the combined segment receives the more restrictive attribute.

For example, if the three partial segments named STACK have relocation attributes UNIT, PAGE, and UNIT, the combined segment has attribute PAGE (i.e., page-aligned). Note that the relocation attribute is applied to the combined segment, not to each component segment. To continue the example, since the relocation is PAGE, the combined segment will start on a page boundary, but the component segments will be packed together without any gaps.

Allocating Memory for Segments

After the segment table is complete, RL51 can locate the segments within the memory spaces. Table 2-1 shows the address spaces used by MCS-51 processors, and the corresponding segment types.

The allocation process has a definite sequence; the exact order is presented in Chapter 3. As an overview, the process follows a general pattern of rules as follows:

1. Each of the types of memory space is allocated independent of the other spaces.
2. Within each space, absolute segments are allocated first, then segments specified within locating controls in the RL51 command, then other relocatable segments.
3. Because the on-chip data space represents three overlapping address spaces (DATA, IDATA, and BIT), the general pattern in rule 2 is modified.

    a. Absolute BIT, DATA, and IDATA segments, and register banks are allocated first.

    b. Segments specified in PRECEDE and BIT controls are allocated next, then other relocatable BIT (and BIT-ADDRESSABLE) segments (following rule 2).

    c. DATA type segments are allocated next: segments in the DATA control first, then other relocatable DATA segments.

    d. IDATA type segments (except ?STACK) are allocated next: segments in the IDATA control first, then other relocatable IDATA segments.

    e. Segments specified in the STACK control are allocated, at as low an address as possible, provided that it is *above* all BIT, DATA, and IDATA segments allocated under (c) and (d).

    f. Last, the segment ?STACK, if it exists and is IDATA, and is not mentioned in an explicit location control, is now allocated, at as low an address as possible, provided that it is *above* all BIT, DATA, and IDATA segments allocated under (c) and (d) and (e).

In most cases, you do not need to use any explicit controls to obtain a satisfactory allocation of segments. RL51 tries to fit your segments into the designated memory spaces as best it can following the rules. As you can see, most of the complexity occurs in the on-chip data space.

Table 2-1. Address Spaces and Segment Types

Memory Space	Maximum Size	Addresses	Segment Type
Code	64K bytes	0000H - 0FFFFH	CODE
External data	64K bytes	0000H - 0FFFFH	XDATA
On-chip data (direct addressing)	128 bytes	00H - 7FH	DATA
On-chip data (indirect addressing)	256 bytes (see 1)	00H - 0FFH	IDATA
Bit space in on-chip data memory	128 bits (see 2)	00H - 7FH	BIT

1. The amount of indirectly addressable on-chip data memory is machine-dependent within the MCS-51 microcomputer family (see the discussion of RAMSIZE control in Chapter 3).

2. This bit space overlaps byte addresses 20H – 2FH in on-chip data memory.

Note: Addresses in the special function register memory (direct data addresses 80H – 0FFH, bit addresses 80H – 0FFH) cannot be relocated; they are always absolute. Thus, these addresses are not referenced in this table.

Rule (f) applies to PL/M-51. PL/M-51 produces for the stack an IDATA segment called ?STACK, whose size is 1. Although, by applying rule (f), RL51 makes the stack as big as possible, it is the user responsibility to ensure that the size of the stack is large enough (the segment map shows where the stack is located).

No rules for the allocation process can guarantee an optimal solution. If you are short of memory and RL51's first try is not satisfactory, you can place the segments in memory using the locating controls. Details on the locating controls are given in Chapter 3.

Overlaying Data Segments

On-chip RAM is a scarce resource on the MCS-51. To economize, the PL/M-51 compiler overlays data segments in the compiled module. RL-51 completes the work by overlaying the data segments across modules. This is accomplished by using the OVERLAY control. If RL-51 informed you about ignored segments due to lack of on-chip RAM, try this control. The use of OVERLAY is, in general, straightforward. However, for complex applications (for example, those with mixed ASM-51 and PL/M-51 modules), consult Chapter 3.

Resolving External References

An external reference points to a location in another module. The EXTERNAL declaration for symbols tells RL51 that the reference is to a location defined in another module. In the latter module, the symbol is declared PUBLIC so that external references to that symbol in other modules can be satisfied.

As it processes the input modules, RL51 builds a table of public symbols and unresolved external references. As each public symbol is added to the table, any external references to that symbol are deleted. After all segments have been located,

the public symbols are bound to absolute addresses. RL51 issues a warning for any unresolved externals that remain in the table.

External symbols and corresponding public symbols must be compatible. That is, both must be defined to address the same address space, or at least one must be defined as a typeless symbol (NUMBER); and if the symbol represents a PL/M-51 procedure name, then both must share the same register bank (i.e., must be declared within the PL/M-51 source modules with the same USING attribute).

Binding Relocatable Addresses

After allocating memory for the combined segments and binding the public symbols, RL51 makes a second pass (pass 2) through the input modules to build the listing file and fixup (i.e., bind to absolute addresses) any relocatable or external references. At this point, RL51 also processes debug records if requested, and performs fixups to any relocatable debug symbols that require processing to compute their absolute addresses.

Introduction

The RL51 program performs two functions for MCS-51 programs:

- The link function, combining a number of object modules specified in an input list into a single object module in an output file
- The locate function, assigning absolute addresses to any relocatable addresses in the input modules

This chapter explains how to enter commands, how to continue a long command onto more than one input line, how to enter comments in the invocation, and how to use abbreviations of the command words.

The chapter then presents a summary of the format of the RL51 invocation command, followed by details on the elements of the command with examples.

RL51 Command Format Summary

Here is a summary of the syntax of the RL51 invocation command. Refer to the Preface for an explanation of the command format notation.

The RL51 command has the overall format:

[*directory* | *device*] R L 5 1 *input-list* [T O *output-file*] [*control-list*]

where

directory | *device*   is the directory or device where RL51 resides.

input-list   is a list of filenames separated by commas. The files named in *input-list* should contain the relocatable modules to be linked and located in the final absolute output module. For each file, you can additionally specify which modules are to be included.

output-file   is the name of the file that is to receive the output module. If you omit this entry, the program will supply a default name based on the first filename in the input list.

control-list   selects options for listing, linking, and locating the output. The listing controls specify what information is to be sent to the listing file, and the page width to be used. The linking controls specify the name of the output module, and determine what debug information is to be placed in the output file. The locating controls allow you to assign absolute addresses to relocatable segments, and to specify the order of relocatable segments within a given type of memory. The configuration control is used to describe the actual configuration the object is aimed to. The overlay control overlays data segments between modules.

The next several sections give details and examples of the elements of the RL51 command. Table 3-1 gives brief definitions of some of the terms used in the controls. A list of abbreviations for command words appears at the end of the chapter.

Table 3-1. Definitions of Common Terms

Term	Definition
name	Names can be from 1 to 40 characters in length and must be composed of letters A - Z, digits 0 - 9, or special characters (?, @, _). The first character must be a letter or a special character.
module-name	Same as name.
segment-name	Same as name.
pathname	A valid filename reference or device reference. See next two items for examples.
filename	A reference to a disk file.
device	A reference to a non-disk device. Examples: :LP:, :CO:, :TO:
value	A 16-bit unsigned integer. Examples: 1011B, 304Q, 4096D (or just 4096), 0C300H
address	Same as value.

Invocation

The RL51 command is a standard operating system invocation. Terminate the command with the RETURN key. Note that the termination carriage return is not shown in the command format notation.

You can continue the invocation line on one or more additional lines by entering the ampersand (&) before you enter the line terminator. The next line then automatically appears with the continuation prompt. Comments can also be entered on the invocation line by placing the comments after the ampersand or semicolon (;) because the compiler ignores all characters that appear after the ampersand or semicolon but before the carriage return/line feed that terminates the line.

Refer to your DOS user's guide for information on submitting batch file commands.

Input List

The input list tells RL51 what files are to be processed. The files must be disk files containing relocatable object modules as described in Chapter 2.

The entry for each file in the list can include the following information:

- The directory or device. If the directory or device is omitted, the default directory or device is assumed.
- The filename. The filename is the name of the object file including an extension if one exists.
- A list of modules enclosed in parentheses. If a module list is provided, only the modules in the list are linked into the output file, and modules not in the list are ignored. If no module list is provided, the default for a non-library file is to link all modules in the file into the output module. The default for a library file is to link only those modules that satisfy previously declared external symbols (see the exact process in Chapter 2 under "Selecting Modules").

If a module named in the module list is not present in the file, the system issues an error message but does not halt the link process.

Module names (specified explicitly or implicitly) must be unique throughout the entire application.

Examples

Following are examples of the RL51 input list—

1. `>RL51 A:prog.obj TO A:prog.abs`

   In this example, the input list has one file (prog.obj in directory A:); RL51 links all the modules in this file into the output file (prog.abs). (For clarity, this and other examples omit the directory in which RL51 resides; the examples assume RL51 resides in the root directory.)

2. `>RL51 a:samp1.obj, a: samp2.obj, a:samp3.obj &`
   `>> TO b:samp.abs`

   In this example, the input list has three files. RL51 links all the modules in each of these files into the output file. (Note that the >> in the second line of the example is generated by the system in response to the continuation character & on the first line of the example.)

3. `>RL51 A:PROG1.OBJ (MOD1, MOD3), A: PROG2.OBJ &`
   `>> (MOD2) TO A:PROG3.ABS`

   Here, the input list has two input files (PROG1.OBJ and PROG2.OBJ). From PROG1.OBJ, only the modules named MOD1 and MOD3 are to be linked into the output file; any other modules in file PROG1.OBJ are ignored by RL51. From PROG2.OBJ, only the module named MOD2 is to be linked.

4. `>RL51 a:plmprg.obj, a:util51.lib, a:io51.lib, &`
   `>> plm51.lib`

   The example introduces a typical linking using libraries. Here, plmprg.obj is linked with two private libraries and with the mandatory library plm51.lib (which must be used if modules generated by plm51 participate in the linkage).

5. `>RL51 a:exampl.obj, cotrig.lib, trig.lib, &`
   `>> cotrig.lib`

   Interaction between libraries (i.e., libraries that reference each other) may sometimes require the same library to be mentioned twice in the input list.

   In the preceding example, cotrig.lib contains the COTANGENT and COSINE trigonometric functions, trig.lib contains the SIN and TANGENT functions, and exampl.obj references the COTANGENT function.

   Because COTANGENT equals 1/TANGENT, trig.lib must be specified to resolve the reference to the TANGENT function. Also, because TANGENT equals SINE/COSINE, cotrig.lib must be respecified to resolve the reference to the COSINE function.

Output File

The output filename is the name of the disk file that is to receive the absolute object module.

If the output file name is omitted, RL51 creates a filename for the output file by removing the extension from the first filename in the input list and using the drive and root name only. If this input file contains no extension, a fatal error occurs. For example, the command:

`RL51 PROG1`

is illegal since the output filename defaults to PROG1.

If there is already a file on the target drive with the name of the output file, that file is overwritten by the new output file.

Examples

Following are examples of RL51 output file—

1. `>RL51 a:prog.obj TO c:prog`

   This example specifies file prog in directory c: as the output file.

2. `>RL51 C:PROG.OBJ`

   This example uses the default output file generated by RL51. The effect is the same as the first example; the output file becomes c:prog.

3. `>rl51 a:sample1.obj, a:sample2.obj TO &`
   `>> c:\myfile\sampl.abs`

   In this example, the output file is in a different directory than the input files, and the directory, filename SAMPL, and the extension .ABS are specified.

Controls

After the output filename, you can add a list of controls to select options for listing, linking, and locating the output. Use blanks (not commas) to separate controls in the list. The same control may not appear more than once in the list; if a duplicate control is encountered, a fatal error results and the program aborts. The next several sections explain the controls and give examples.

Listing Controls

The listing file output by RL51 can contain a link summary, a symbol table, an IXREF report, and a list of error messages. The link summary can contain a memory map of the linked segments.

The listing controls are the PRINT option, the PAGEWIDTH control, the MAP option, the SYMBOLS option, the PUBLICS option, the LINES option, and the IXREF option. These controls allow you to specify the file or device to receive the output listing, to omit the listing file altogether, to omit the map from the link summary, or to omit local symbols, public symbols, or line numbers from the symbol table. You may also specify if you wish to have the IXREF report generated, and the specific page width to be used.

NOTE

The information in the listing file is taken from the input object modules. If these are generated without the DEBUG option, the SYMBOLS, PUBLICS, and LINES information will not be available for listing.

PRINT/NOPRINT

The print options control the destination of the list file.

To direct the list file to a disk file, the print control format is

`PRINT ( [directory/device] filename [ .ext]  )`

Example

`>RL51 a:sample1.obj &`
`>> print (a:sample.lst)`

To direct the list file to a device other than a disk file, the print control format is

`PRINT ( :device: )`

where

    device   is a device code. Common devices are CO (console), LP (line printer), TO (terminal other than console), and VO (video terminal screen).

If you omit the print control, or if you enter the command word PRINT without a filename or device name, RL51 creates a disk file for the listing. The name of the default listing file has the same root as the output filename and has an extension of M51; the drive number is also the one used in the output filename.

Example

`>RL51 A:PROG.OBJ, A:PROG1.OBJ TO B:PROG2.ABS`

The output listing filename may not be the same as the output filename or any of the filenames in the input list. If the listing file duplicates an input or output filename, a fatal error results. If the listing filename already exists on the target directory, the old file with that name is overwritten by the new listing file.

The NOPRINT option specifies that no output listing file is to be produced. NOPRINT overrides the MAP, SYMBOLS, PUBLICS, LINES and IXREF controls.

PAGEWIDTH

The PAGEWIDTH control specifies the maximum number of columns per line in the print output file. The control takes the form

`PAGEWIDTH(width)`

where

    width   is an unsigned number which specifies the maximum page width to be used.

The allowable range for *width* is 72 to 132. The default PAGEWIDTH is 78.

Listing Switches

The MAP, SYMBOLS, PUBLICS, LINES and IXREF controls select what portions of the listing files are to be generated. The default of any switch (with the exception of IXREF) is the positive form (MAP, SYMBOLS, PUBLICS, and LINES). Table 3-2 summarizes the listing switches.

IXREF/NOIXREF

This control specifies whether or not to produce the inter-module cross reference report. If IXREF is specified, the report is appended to the print file.

A selection list may be added to the positive form (only) of the IXREF control. A selection list causes RL51 to output or suppress output of various selected entries to

Table 3-2. Listing Switches

Switch	Effect
MAP	Output memory map to link summary
NOMAP	Suppress memory map
SYMBOLS	Output local symbols to symbol table
NOSYMBOLS	Suppress local symbols
PUBLICS	Output public symbols to symbol table
NOPUBLICS	Suppress public symbols
LINES	Output line numbers to symbol table (high-level language transla-tors only)
NOLINES	Suppress line numbers
IXREF	Append intermodule cross-reference report to print file
NOIXREF	Suppress the intermodule cross-reference report

the IXREF report. An entry consists of a symbol and a module where this symbol is referenced (either as public or as external). The general form of the IXREF control is

```
IXREF [(selection-item [,...])]
```

where

selection-item   is either (NO)GENERATED or (NO)LIBRARIES. If IXREF is specified and any of the selection items are omitted, the missing selection item assumes its positive form. A selection item may appear at most once.

The selection-items are best explained by describing the effect of their negative form.

The NOGENERATED control causes RL51 to surpress output of entries whose symbol name begins with a question mark (?); such symbols are usually PL/M-51 generated symbols. The GENERATED form of the control causes RL51 to output such entries also.

The NOLIBRARIES control causes RL51 to surpress output of entries whose module resides within a library. The LIBRARIES form of the control causes RL51 to include all libraries in the IXREF report.

The selection list is used to control the number of entries collected for the IXREF report. This is needed when an excessive number of IXREF entries make it impossible for RL51 to generate the IXREF report.

Examples

1. >RL51 a:prog.obj nosymbols nopublics nolines

    Because the default for any listing switch (except ixref) is the positive form, the main use of the switches is to suppress unwanted information. The invocation given in this example will suppress the entire symbol table.

2. >rl51 a:prog.obj print (a:prog.m51) nomap nosb &
    >> noli

    In this example, only the public symbols will be printed (no map or other symbols or lines). Note the use of abbreviations (nosb for nosymbols and noli for nolines) to save keystrokes. A complete list of abbreviated forms appears at the end of

Chapter 3. Note that the blank separating print from its parameters is optional; you could also use print(a:prog.m51).

3. >RL51 a:prog.obj, a:procs.obj, a:plm51.lib &
    >> ixref(nogn)

    This example suppresses generated symbols from the ixref report. Using the nogn (nogenerated) selection item prevents PL/M-51 run-time library procedures from being written to the ixref report.

Linking Controls

The linking controls allow you to name the resultant output module and to specify which debug information is to be copied to the output module.

NOTE

In order to obtain the debug information (SYMBOLS, PUBLICS, or LINES), the DEBUG control must be included in the invocation line for the translator used to produce the input modules.

NAME

The NAME control allows you to name the output module. The format is

```
NAME (module-name)
```

If the NAME control is not used, the output module-name defaults to the name of the first input module processed.

Example

```
>RL51 a:sampl1.obj, a:sampl2.obj TO a:sample.abs &
>> name(SAMPLE_PROGRAM)
```

In this example, the name SAMPLE_PROGRAM is assigned to the output module. Note that the blank between NAME and its parameter is optional and can be omitted.

Linking Switches

The DEBUGSYMBOLS, DEBUGPUBLICS, and DEBUGLINES controls select what kinds of debug information are to be included in the output file. The default of any switch is always the positive form (DEBUGSYMBOLS, DEBUGPUBLICS, and DEBUGLINES). Table 3-3 summarizes the linking switches.

Examples

1. >RL51 a:prog1.obj nodebugsymbols nodebuglines

    Because the linking switches default to the positive form, you will usually use the negative forms to suppress unwanted debug information in the output file. In this example, the output file contains only the information for the public symbols.

2. >RL51 a:prog1.obj nodp nodl

    In this example, only the local symbols are output to the absolute file. Note the use of abbreviations (nodp for nodebugpublics and nodl for nodebuglines).

Table 3-3. Linking Switches

Switch	Effect
DEBUGSYMBDLS	Copies local symbol information to output file
NODEBUGSYMBDLS	Suppresses local symbols
DEBUGPUBLICS	Copies public symbol information to output file
NDDEBUGPUBLICS	Suppresses public symbols
DEBUGLINES	Copies line number information (high-level language translators only) to output file
NDDEBUGLINES	Suppresses line numbers

Locating Controls

The locating controls allow you to assign absolute addresses to relocatable segments, to specify the ordering of relocatable segments of a given type in memory, and to force allocation of segments into a specifie range of addresses.

Allocation Sequence

The system allocates memory in accordance with segment attributes and locating controls, using a fixed order of precedence. The precedence of the allocating operations (grouped by type of memory space) is as follows:

Internal Data Space:

- Absolute BIT, DATA, and IDATA segments, and register banks
- Segments specified in a PRECEDE control in the RL51 command
- Segments specified in a BIT control in the RL51 command
- DATA type segments with relocation equal to BIT-ADDRESSABLE
- Other relocatable bit segments
- Segments specified in a DATA control in the RL51 command
- DATA type segments with relocation equal to UNIT-aligned
- Segments specified in an IDATA control in the RL51 command
- Other relocatable IDATA segments, except ?STACK
- Segments specified in a STACK control in the RL51 command
- ?STACK, if it is IDATA and has not been specified in any other locate control

External Data Space:

- Absolute external data segments
- Segments specified in an XDATA control in the RL51 command
- Other relocatable external data segments

Code Space:

- Absolute code segments
- Segments specified in a CODE control in the RL51 command
- Other relocatable code segments

NOTE

In most cases, the allocation algorithm will produce a workable solution without requiring the user to enter any locating controls in the RL51 command. These controls are intended for the experienced user, in cases where running RL51 without them does not give a good enough result.

Format Summary

The locating controls have the format

control (*segment* [, . . .])

where

    segment : = *segment-name* [(*base-address*)]

The segments specified in the locating controls are allocated in the order they appear; the first segment is assigned the lowest possible address, and succeeding segments receive higher and higher addresses.

The user has the option of specifying the base address of any or all segments. Segments with specified base addresses must appear in the list in ascending numerical order. Segments named in a locating control with a specifie base address are allocated at that address irrespective of segment overlap or segment type contradiction, as long as ascending order is maintained. Base addresses are byte addresses except for the BIT locating control, where addresses are bit addresses in the bit space(0 to 127).

Table of Locating Controls

Table 3-4 lists the locating controls in order of precedence. The first column gives the name of the control. The second column describes the address space affected by the control. The third column gives the address range for segments within each control. The last column shows what types of segments are allowed for each control; for each valid type, the column also shows the allowable relocation attributes. (Refer to the *MCS-51 Macro Assembler User's Guide* and *PL/M-51 User's Guide* for details on segment types and relocation attributes.)

Notes On Locating Controls

The following notes refer to table 3-4.

1. Bit addresses for non-BIT segments in the BIT control must be on byte boundaries; that is, they must be divisible by eight. (BIT-type segments can be aligned on bit boundaries.)

Table 3-4. Locating Controls

Control	Address Space	Address Range (Hex)	Segment Types (end Attributes)
PRECEDE	Register banks and bit-addressable space in on-chip data RAM	00H-2FH	DATA (UNIT-aligned); IDATA
BIT	Bit-addressable space in on-chip data RAM	00H - 7FH (see note 1)	BIT; DATA; IDATA
DATA	Directly-addressable on-chip data RAM	00H - 7FH	DATA (UNIT-aligned); IDATA
IDATA	Indirectly-addressable on-chip data RAM	00H - 0FFH (see note 2)	IDATA
STACK	Same as IDATA (see note 3)	Same as IDATA	Same as IDATA
XDATA	External data RAM	0 - 0FFFFH	XDATA
CODE	Code memory	0 - 0FFFFH	CODE

2. The range of addresses for the IDATA control is dependent on the target machine. See the RAMSIZE control later in this chapter.

3. The STACK control specifies which segments are to be allocated uppermost in the IDATA space. The memory accessed starts after the highest on-chip RAM address occupied by any previously allocated segment and continues to the top of the IDATA space.

NOTE

This control has no other effect on any segments.

The IDATA ?STACK segment, if it exists, is placed higher than segments that were mentioned in the STACK control.

The STACK control provides a convenient way to handle the stack (usually for ASM51-based application, where ?STACK is not used).

First, assign the stack pointer (SP) to a relocatable segment; consider the following ASM51 example:

```
STACK_AREA SEGMENT IDATA        ; SEGMENT directive in source.
        DS      10H             ; Reserve 16 bytes for stack.

........                        ; Other CODE instructions.

        MOV     SP, #STACK_AREA-1  ; Initialize SP.
```

Then, at relocation time, specify the segment named STACK_AREA in a STACK locating control:

```
RLS1 ... STACK (STACK_AREA)
```

where

ellipsis (...)   represents the rest of the invocation line exclusive of the STACK control.

NOTE

If the application contains modules produced by PL/M-51, the ?STACK should be used as the stack segment.

Examples

1. `>RLS1 A:PROG1.OBJ, A:PROG2.OBJ TO A:PROG.ABS &`
   `>> PRECEDE (MESSAGE1) XDATA (ARRAY1 (2S6), &`
   `>> ARRAY2 (S12))`

   In this example, the DATA (or IDATA) segment names MESSAGE1 will be allocated space in on-chip RAM in the lowest available location, overlapping the BIT space if necessary. The XDATA control specifies that the two arrays are to be located at specific addresses (e.g., for debugging).

2. `>RLS1 A:TEST.OBJ STACK (STACK_AREA)`

   Here, the STACK control allocates the uppermost portion of IDATA space for the segment named STACK_AREA.

3. `>RLS1 APROG.OBJ, BPROG.OBJ, PLMS1.LIB &`
   `>> CODE (MOD1 (4000H), MOD2, MOD3)`

   Here, the CODE control allocates space in code memory for segments MOD1, MOD2, and MOD3. MOD1 is aligned at location 4000H. MOD2 and MOD3 are assigned contiguous addresses after MOD1.

Configuration Controls

The configuration controls are used to describe the actual configurations that objects are aimed to.

This group contains the RAMSIZE control.

RAMSIZE

The RAMSIZE control format

`RAMSIZE (value)`

where

value   is a number in the range 128 to 255.

RAMSIZE specifies the maximum amount of on-chip RAM that may be allocated for the user program. The default value for RAMSIZE is 128 (as is the case for the 8051). If the object is aimed at more than one configuration of the MCS-51 family, specify the MINIMUM of all on-chip RAM sizes among all machines you want to link.

The sole use of this control is to enable RL51 to check on-chip memory size constraints at RL-time and thus avoid confusion at ICE-time.

OVERLAY/NOOVERLAY Controls

The linker allows overlaying of on-chip RAM segments among modules, under the specification of the OVERLAY control. Two segments can be overlaid if all the following conditions exist:

* The segments have the same type (DATA, IDATA, BIT, or BITADDRESSABLE).

* The segments use the same register bank (determined by the USING attribute or the REGISTERBANK control).

* The segments are marked as overlayable. Currently, this is done only by the PL/M-51 compiler. ASM51 (V2.1 and lower) lacks this feature. Therefore, assembler segments are considered non-overlayable.

* The segments belong to disjoint modules. That is, no procedure in one module can directly or indirectly call a procedure from the other.

The default is NOOVERLAY. No overlaying of on-chip RAM segments is done by the linker.

The general form of the OVERLAY control is as follows:

`OVERLAY [ (overlay-unit [ , ... ] ) ]`

where

overlay-unit   is ov-module-name calls ov-module-name.

ov-module-name   is a legal RL51 module name or *, which stands for all the module names.

calls   is > or].

OVERLAY

If the OVERLAY control appears in the invocation line without arguments, the linker assumes that no intra-module calls exist except for those deducible from the PUBLIC-EXTERNAL declarations, and that overlaying of all overlayable segments is safe.

NOOVERLAY

The linker does not overlay data segments.

OVERLAY (A > B) or (A] B)

If the OVERLAY control appears in the invocation line with arguments, it indicates that there are invisible calls between modules. In the OVERLAY control syntax, either the greater than sign (>), or the right square bracket ( ]) may be used in the calls relationship. The greater than sign will be used in the text. The notation A > B means that module A calls module B. In this case, the linker overlays all overlayable segments, except that segments from A are not overlaid by segments from B. Note that the added connection can prevent other segments from overlaying. For example, if the segment A was overlaid with the segment D, and B calls D (visibly by PUBLIC-EXTERNAL declarations), then the effect of A > B is that A and D will not be overlaid, since A can call D through B.

OVERLAY (A > *, * > B) or (A] *, *] B)

A module can be declared as non-overlayable in two ways. The argument A > * indicates that the module A calls all other modules. On the other hand, * > A means every module calls A. In either case, no segments from A will be overlaid. The effect of each form depends on the nature of A. For example, if the * > A form is used and A visibly calls all other modules, then every module can call (through A) each other module. In this case, the linker will not perform any overlays.

The overlaying of data segments in on-chip RAM has the following restrictions:

- The OVERLAY control cannot be invoked with the IXREF selection items NOGENERATED or NOLIBRARIES. RL51 generates an error if either one is specified.
- Combined segments and segments appearing in locating controls are not overlaid by the linker.

Following is an example in which two disjoint modules share the same on-chip RAM area:

```
mod1: DO;

    THREE_BEARS: PROCEDURE PUBLIC;
        DECLARE LITTLE_BEARS_BED BYTE;
        IF BOOLEAN (LITTLE_BEARS_BED) THEN
            CALL MSG(.('SOMEONE''S BEEN IN MY BED!'),0);
        LITTLE_BEARS_BED = 0;
    END THREE_BEARS;

END mod1;
```

```
mod2: DO;

    GOLDILOCKS: PROCEDURE PUBLIC;
        DECLARE SPARE_BED BYTE;
        SPARE_BED = 1;
    END GOLDILOCKS;

END mod2;

main_story: DO;

    THREE_BEARS: PROCEDURE EXTERNAL; END;
    GOLDILOCKS: PROCEDURE EXTERNAL; END;

    CALL THREE_BEARS;
    CALL GOLDILOCKS;
    CALL THREE_BEARS;

END main_story;
```

In this example, the linker reserves the right to use the LITTLE_BEARS_BED as a SPARE_BED because the two procedures are never active simultaneously.

To perform overlaying, the linker must determine which procedures are active simultaneously. To do this, the linker assumes that all CALLs can be executed. For example, if procedure A calls procedure B, and B calls procedures C and D, then the linker can overlay RAM variables from C only with the RAM variables of D.

The linker, however, looks only at the PUBLIC-EXTERNAL declarations. It assumes that any reference to an EXTERNAL procedure will be executed, but ignores the possibility of hidden calls. The arguments to the OVERLAY control are therefore needed to specify those interconnections between modules that cannot otherwise be detected by the linker.

Such situations arise if the interconnection is done by a computed call to an external procedure whose address is not determined by a simple PUBLIC-EXTERNAL relationship. For example, module A imports from module B a public variable that contains the address of a local or public procedure in B. Module A then performs a computed call to the procedure in B. The rule can be stated as follows: The linker assumes a connection from module A to module B if there exists an external reference in A to a public procedure in B. In all other cases, hidden connections must be explicitly given as arguments to the OVERLAY control.

Following is an example of a computed call to an external procedure:

```
MOD1: DO;

    DECLARE I_O_CLEAR WORD EXTERNAL;
    . . .
    CALL I_O_CLEAR;
    . . .
END MOD1;
```

In another module, you have:

```
MOD2: DO;

        DECLARE I_O_CLEAR WORD PUBLIC;

        READER: PROCEDURE;

                I_O_ERROR: PROCEDURE;
                . . .
                END I_O_ERROR;

                I_O_SUCCESS: PROCEDURE;
                . . .
                END I_O_SUCCESS;

                IF ERR_CODE <> 0
                        THEN I_O_CLEAR = .I_O_ERROR;
                        ELSE I_O_CLEAR = .I_O_SUCCESS;
                . . .

        END READER;

END MOD2;
```

In the above procedure, MOD1 invokes a procedure defined in MOD2. To prevent the linker from overlaying on-chip RAM variables of MOD2 with on-chip RAM variables of MOD1, the following form of the OVERLAY control must be used:

```
OVERLAY (MOD1 > MOD2)
```

Overlaying can be a good way of economizing on-chip RAM space; however, overlaying may, in some cases, give worse results. For example, if most procedures call one another, the resulting segments will expand, making it more difficult for the linker to allocate a few large segments than many small ones.

The outcome of the overlaying process can be checked by inspecting the link map. All overlaid segments are indicated by **OVERLAP**. Warning (4), DATA SPACE MEMORY OVERLAP, is not generated for those segments.

Abbreviations for Command Words

Most of the command words in the RL51 command have short forms to save you keystrokes over the full spellings. Here is a list of the command words and their abbreviations.

Command Word	Abbreviation
BIT	BI
CODE	CO
DATA	DT
DEBUGLINES	DL
DEBUGPUBLICS	DP
DEBUGSYMBOLS	DS
GENERATED	GN
IDATA	ID
IXREF	IX
LIBRARIES	LB

LINES	LI
MAP	MA
NAME	NA
NODEBUGLINES	NODL
NODEBUGPUBLICS	NODP
NODEBUGSYMBOLS	NODS
NOGENERATED	NOGN
NOIXREF	NOIX
NOLIBRARIES	NOLB
NOLINES	NOLI
NOMAP	NOMA
NOOVERLAY	NOOL
NOPRINT	NOPR
NOPUBLICS	NOPL
NOSYMBOLS	NOSB
OVERLAY	OL
PAGEWIDTH	PW
PRECEDE	PC
PRINT	PR
PUBLICS	PL
RAMSIZE	RS

The RL51 program produces three outputs: console displays, a listing file, and the absolute object module file. This chapter describes these outputs and gives examples. As discussed in Chapter 3, the listing controls in the RL51 command allow the user to suppress some information in the listing file, and the linking controls can suppress some information in the absolute object file.

Console Display

The console displays produced by RL51 consist of a sign-on message and any error messages that occur. The sign-on is as follows:

system-id MCS-51 RELOCATOR AND LINKER Vx.y

where

 x.y   is the version number.

Listing File

RL51 produces a listing file unless it is suppressed in the RL51 invocation. The RL51 listing file contains:

- A summary of the link and locate process
- A symbol table, as specified in the RL51 invocation
- An inter-module cross-reference listing (IXREF)
- Error messages detected by RL51

Link Summary

A sample of a link summary is shown in figure 4-1. The summary includes the following kinds of information:

- A header echoing the RL51 invocation.
- Input modules included in the link process. Input modules are identified by module name and file name.
- A link map (unless suppressed by the NOMAP control). The map lists all allocated segments, giving the type, base address, and length of each segment. The map also identifies segment overlaps and gaps in the memory space.
- A list of segments that were ignored in the link process. If any segments were ignored, the reasons for doing so will be reported later as an error.
- A list of unresolved external symbols. An external symbol is unresolved when it is not matched by a public symbol in one of the input modules. Each occurrence of an unresolved external symbol in a module will be reported later as an error.
- A list of all symbols that were ignored in the locate process. A symbol is ignored when the same name appears as a public symbol in different modules, or has attributes that are incompatible with external references, or belongs to an ignored segment. Each occurrence of an ignored symbol in a module will be reported later as an error.

```
system-id MCS-51 RELOCATOR AND LINKER, Vx.y INVOKED BY:
RLS1 FILE1.EXT(MOD1,MOD2); FILE2.EXT TO OUTFIL.EXT &
NAME (EXAMPLE) MAP PRIHT (:LP:)

INPUT MODULES INCLUDED
   FILE1.EXT(MOD1)
   FILE1.EXT(MOD2)
   FILE2.EXT(MOD3)

LINK MAP FOR OUTFIL.EXT(EXAMPLE)

               TYPE     BASE      LENGTH    RELOCATION    SEGMENT NAME
               ----     ----      ------    ----------    ------------
               REG      0000H     0008H                   "REG BANK 0"
               DATA     0008H     0010H     UNIT
               DATA     0014H     0008H     ABSOLUTE       DATA_SEG_1
**OVERLAP**    REG      0018H     0008H                   "REG BANK 3"
               BIT      0020H     0001H.6   UNIT          A_BIT_SEG
                        0021H.6   0000H.2                 ***GAP***
               DATA     0022H     0001H     BITADDR       DATA_SEG_2
               DATA     0023H     000BH     ABSOLUTE
               IDATA    002EH     0042H     UNIT          STACK_SEG
                        0070H     0010H                   ***GAP***

               XDATA    0000H     C000H     UNIT          DYNAMIC_MEM

               CODE     0000H     1389H     UNIT          PROC1
                        1389H     0477H                   ***GAP***
               CODE     1800H     07A5H     INBLOCK       PROC2

IGNORED SEGMENTS
   DYNAMIC POOL

UNRESOLVED EXTERNAL SYMBOLS
   INVERT

IGNORED SYMBOLS
   BIT256
```

Figure 4-1. Link Summary

NOTE

1. For bit addresses, the display format is *byte-address.bit-address* (example: 0020H.7 for bit 7 of byte 0020H). However, when bit 0 of a byte is referenced, only the byte address is displayed (the .0 is not displayed).

2. References to an unresolved external symbol, an external symbol referring to an ignored public symbol, or a reference to an ignored segment will produce additional error messages.

Symbol Table

The listing file contains a symbol table as specified by the SYMBOLS, PUBLICS, and LINES controls in the RL51 invocation. A sample symbol table is shown in figure 4-2.

```
SYMBOL TABLE FOR OUTFILE.EXT(EXAMPLE)

VALUE            TYPE             NAME
-----            ----             ----

-------          MODULE           MEMRY
D:0032H          PUBLIC           LOW_MEM_PTR
B:0020H          PUBLIC           INIT_FLAG
B:0020H.1        PUBLIC           FULL_FLAG
D:0034H          PUBLIC           HIGH_MEM_PTR
X:0000H          PUBLIC           DYNAMIC_MEMORY
-------          PROC             ALLOCATE
D:0064H          SYMBOL           NUM_BYTES
D:0066H          SYMBOL           POOL_SELECTOR
D:0068H          SYMBOL           ALLOC_PTR
B:0020H.2        SYMBOL           FLAG
C:0000H          LINE#            19
C:007H           LINE#            20
C:0010H          LINE#            21
C:0013H          LINE#            22
------           DO
D:006AH          SYMBOL           I
C:0018H          LINE#            23
C:0021H          LINE#            24
C:0028H          LINE#            2S
C:002FH          LINE#            26
C:0032H          LINE#            27
-------          ENDDO
C:0037H          LINE#            28
C:0040H          LINE#            29
C:004FH          LINE#            30
C:0057H          LINE#            31
C:005FH          LINE#            32
C:0068H          LINE#            33
C:006FH          LINE#            34
C:0076H          LINE#            35
C:0082H          LINE#            36
C:008FH          LINE#            37
C:0094H          LINE#            38
-------          ENDPROC          ALLOCATE
-------          ENDMOD           MEMRY
```

Figure 4-2. Symbol Table

NOTE

The information in the listing file is taken from the input object modules. If these are generated without the DEBUG option, the SYMBOLS, PUBLICS, and LINES information will not be available for listing.

The symbol table contains scope definitions and information about the symbols and line numbers. Scope definition identifies the module, DO block or procedure that contains the symbol or line number. Note that when the table contains only public symbols (i.e., NOSYMBOLS and NOLINES controls are in effect), scope definition is by module only.

Each entry in the table consists of three parts, as follows:

- VALUE. The value is the absolute address of the symbol. The address is prefixed with a letter indicating the type of address space (C, code; D, internal data; I, indirect internal data; B, bit space; X, external data; N, typeless number). A byte address (or a bit address on a byte boundary) is shown as a four-digit hexadecimal number (example: 00E0H). A bit address (unless it is on a byte boundary) is shown as a byte address followed by a period and the bit offset (1 through 7) into the byte.
- TYPE. The type field identifies the entry as a local symbol (SYMBOL), a public symbol (PUBLIC), segment (SEGMENT), or a line number (LINE#).
- NAME. The name field gives the name of the symbol, or the number of the line.

For scope definition, a line is printed for the beginning and end of each block. The TYPE field shows the type of block (MODULE, DO, or PROC for PROCEDURE), and the end of each block (ENDMOD, ENDDO, ENDPROC). The NAME field shows the name of the block, if any.

NOTE

Line number information and scope definitions other than MODULE are applicable only to object files produced by high-level language translators (e.g., PL/M-51).

Inter-Module Cross-Reference Report (IXREF)

The listing file contains an IXREF report as specified by the IXREF control and its associated selection list in the RL51 invocation. A sample IXREF report is shown in figure 4-3.

The IXREF report consists of an alphabetically sorted list of symbols. Each such symbol begins a new line and represents a symbol that was declared as PUBLIC or EXTERNAL in at least one of the input modules. Each symbol is followed by its corresponding address space, followed by a semicolon. To the right of the semicolon starts a list of modules in which the symbol was declared PUBLIC or EXTERNAL. The first module name in the list is the one in which the symbol was declared PUBLIC. If a symbol is unresolved, or if a symbol is defined in a library and the NOLIBRAR-IES selection item is in effect, then the string ** UNRESOLVED ** appears in front of the modules list.

Error Messages

RL51 displays error messages on the console and copies them to the end of the listing file unless the listing file is suppressed.

RL51 error messages describe warnings, errors, and fatal errors. A warning is a detected condition that may or may not be what the user desired; a warning does not terminate the link/locate operation. An error does not terminate operation, but probably results in an output module that cannot be used. A fatal error terminates operation of RL51.

Refer to Appendix B for a list of the error messages and probable causes.

```
INTER-MODULE CROSS-REFERENCE LISTING
---------------------------------------

NAME. . . . . . . . . . USAGE     MODULE NAMES
---------------------------------------------

?CHECK_EQ?BYTE. . . . DATA;   CHKEQ   TESBAS
?CHECK_EQ_BITS?BIT. BIT;      CHKEQ
?P0008. . . . . . . . CODE;   ?P0008   TESBAS
?P0015. . . . . . . . CODE;   ?P0015   TESBAS
?P0016. . . . . . . . CODE;   ?P0016   TESBAS
?PINOR. . . . . . . . CODE;   TESBAS   ?PIVOR
?PIVOR. . . . . . . . CODE;   ?PIVOR   TESBAS
?PSWOR. . . . . . . . NUMB;   TESBAS   ?PIVOR
CHECK_EQ. . . . . . . CODE;   CHKEQ    TESBAS
CHECK_EQ_BITS . . . CODE;     CHKEQ
CHECK_EXIT. . . . . CODE;     CHKEQ
CHECK_INIT. . . . . CODE;     CHKEQ
PUB00 . . . . . . . . CODE;   MODULE_0   MODULE_1   MODULE_2
PUB01 . . . . . . . . CODE;   MODULE_0   MODULE_1   MODULE_2
PUB02 . . . . . . . . CODE;   MODULE_0   MODULE_1   MODULE_2
PUB03 . . . . . . . . CODE;   ** UNRESOLVED **   MODULE_1   MODULE_2
PUB04 . . . . . . . . CODE;   MODULE_0   MODULE_1   MODULE_2
PUB05 . . . . . . . . CODE;   MODULE_0   MODULE_1   MODULE_2
PUB06 . . . . . . . . CODE;   ** UNRESOLVED **   MODULE_1   MODULE_2
PUB07 . . . . . . . . CODE;   MODULE_0   MODULE_1   MODULE_2
PUB08 . . . . . . . . CODE;   MODULE_0   MODULE_1   MODULE_2
PUB09 . . . . . . . . CODE;   MODULE_0   MODULE_1   MODULE_2
PUB10 . . . . . . . . CODE;   MODULE_0
PUB11 . . . . . . . . XDATA;  MODULE_0
PUB12 . . . . . . . . DATA;   MODULE_0
PUB13 . . . . . . . . IDATA;  MODULE_0
PUB14 . . . . . . . . BIT;    MODULE_0
PUB15 . . . . . . . . NUMB;   MODULE_0
PUB16 . . . . . . . . CODE;   MODULE_0
PUB17 . . . . . . . . CODE;   MODULE_0
PUB18 . . . . . . . . CODE;   MODULE_0
PUB19 . . . . . . . . CODE;   MODULE_0
PUBX0 . . . . . . . . CODE;   MODULE_1   MODULE_0
PUBX1 . . . . . . . . CODE;   MODULE_1   MODULE_0
PUBY0 . . . . . . . . CODE;   ** UNRESOLVED **   MODULE_1   MODULE_2
PUBZ0 . . . . . . . . CODE;   ** UNRESOLVED **   MODULE_1
```

Figure 4-3. IXREF Listing

Absolute Object File

The linking and locating process combines one or more relocatable object files into one absolute object file. The absolute object file contains one module; the absolute module consists of

- A module header record that identifies the module.
- A set of intermixed content and debug records. The content records contain the program code. The debug records contain the location and scope of local symbols, public symbols, segment symbols, and line numbers, as specified by the DEBUG-SYMBOLS, DEBUGPUBLICS, and DEBUGLINES controls in the RL51 invocation.
- A module end record that verifies the module name.

This chapter shows three brief examples of program development using ASM51, PL/M-51, and RL51. The first example is the sample program discussed in the *ASM51 User's Guide*; the example shows how to assemble each of the three modules, then link and locate them into a single absolute object module with RL51. The second example is a short program that illustrates the use of the locating controls. The third example shows the use of RL51 with PL/M-51 modules, emphasizing the library process.

Using Multiple Modules

The first example is a program of three modules, named SAMPLE, CONSOLE_IO, and NUM_CONVERSION. The source for these modules is in three files, SAMP1.A51, SAMP2.A51, and SAMP3.A51, respectively. To assemble these modules, invoke the assembler as follows:

```
ASM51 SAMP1.A51 DEBUG

ASM51 SAMP2.A51 DEBUG

ASM51 SAMP3.A51 DEBUG
```

Note that this example assumes the three source files are on the same directory or device as the assembler and linker/locator, and that the output file will be sent to the same directory or device. The assembler invocations use the DEBUG control to have the symbol tables output to the object files for the three modules.

After assembly is complete, the system has created object files SAMP1.OBJ, SAMP2.OBJ, and SAMP3.OBJ, and listing files SAMP1.LST, SAMP2.LST and SAMP3.LST. The three listing files are shown in figures 5-1, 5-2, and 5-3.

To link and locate the three modules, enter the command

```
RL51 SAMP1.OBJ, SAMP2.OBJ, SAMP3.OBJ &
**TO SAMPLE &
**PRINT (SAMPLE.LST) SYMBOLS LINES PUBLICS
```

After the RL51 program has executed, the system has placed the absolute object module in file SAMPLE, and an output file with information on the link and locate process in file SAMPLE.LST. The output file also contains symbol table information as requested by the SYMBOLS, LINES, and PUBLICS controls. The listing file is shown in figures 5-1 through 5-3; figure 5-4 shows the output file.

```
MCS-51 MACRO ASSEMBLER SAMPLE
systemid MCS-51 MACRO ASSEMBLER Vx.y
OBJECT MODULE PLACED IN SAMP1.OBJ
ASSEMBLER INVOKED BY: ASMS1 SAMP1.AS1 DEBUG

LOC   OBJ          LINE   SOURCE

                    1        NAME SAMPLE
                    2      ;
                    3         EXTRN code (put_crlf, put_string, put_data_str)
                    4         EXTRN code (get_num, binasc, ascbin)
                    5      ;
                    6         CSEG
                    7      ; This is the initializing section. Execution
                    8      ; always starts at address 0 on power-up.
0000                8        ORG  0
0000 758920        10        MOV  TMOD,#001000008   ; Set timer to auto-reload
0003 758D03        11        MOV  TH1,#(-253)       ; Set timer for 110 8AUD
0006 75980A        12        MOV  SCON,#110110108   ; Prepare the Serial Port
0009 028E          13        SET8 TR1               ; Start clock
                   14      ;
                   15      ; This is the main program. It's an infinite loop,
                   16      ; where each iteration prompts the console for 2
                   17      ; input numbers and types out their sum.
                   18        START:
                   19      ; Type message explaining how to correct a typo
0008 900000  F     20        MOV  DPTR,#typo_msg
000E 120000  F     21        CALL put_string
0011 120000  F     22        CALL put_crlf
                   23      ; Get first number from console
0014 900000  F     24        MOV  DPTR,#num1_msg
0017 120000  F     25        CALL put_string
001A 120000  F     26        CALL put_crlf
001D 7800          27        MOV  R0,#num1
001F 120000  F     28        CALL get_num
0022 120000  F     29        CALL put_crlf
                   30      ; Get second number from console
0025 900000  F     31        MOV  DPTR,#num2_msg
0028 120000  F     32        CALL put_string
002B 120000  F     33        CALL put_crlf
002E 7800          34        MOV  R0,#num2
0030 120000  F     35        CALL get_num
```

Figure 5-1. SAMP1 Listing File

```
0033 120000  F     36        CALL put_crlf
                   37      ; Convert the ASCII numbers to binary
0036 7900    F     38        MOV  R1,#num1
0038 120000  F     39        CALL ascbin
003B 7900    F     40        MOV  R1,#num2
003D 120000  F     41        CALL ascbin
                   42      ; Add the 2 numbers, and store the results in SUM
0040 E500    F     43        MOV  a,num1
0042 2500    F     44        ADD  a,num2
0044 F500    F     45        MOV  sum,a
                   46      ; Convert SUM from binary to ASCII
0046 7900    F     47        MOV  R1,#sum
0048 120000  F     48        CALL binasc
                   49      ; Output sum to console
004B 900000  F     50        MOV  DPTR,#sum_msg
004E 120000  F     51        CALL put_string
0051 7A04          52        MOV  R2,#4
0053 7900          53        MOV  R1,#sum
0055 120000  F     54        CALL put_data_str
0058 8081          55        JMP  start
                   56      ;
----               57            OSEG  at 8
0008               58        STACK:  DS  8   ; At power-up the stack pointer is
                   59                        ; initialized to point here.
                   60      ;
                   61        DATA_AREA      segment DATA
                   62        CONSTANT_AREA  segment CODE
                   63      ;
----               64        RSEG data_area
0000               65        NUM1: DS  4
0004               66        NUM2: DS  4
0008               67        SUM:  DS  4
                   68      ;
----               69        RSEG constant_area
0000 5459504S      70        TYPO_MSG: DB 'TYPE X TO RETYPE A NUMBER',00H
0004 20SES820
0008 S44F2052
000C 4SS4S950
0010 45204120
0014 4ES54D42
0018 4SS2
001A 00
```

Figure 5-1. SAMP1 Listing File (Cont'd.)

```
001B 54595045   71        NUM1_MSG: DB 'TYPE IN FIRST NUMBER: ',00H
001F 20494E20
0023 46495253
0027 54204E55
002B 4D424552
002F 3A20
0031 00
0032 54595045   72        NUM2_MSG: DB 'TYPE IN SECOND NUMBER: ',00H
0036 20494E20
003A 5345434F
003E 4E442041
0042 554D4245
0046 523A20
0049 00
004A 54484520   73        SUM_MSG:  DB 'THE SUM IS ',00H
004E 53554D20
0052 495320
0055 00
                 74        i
                 75        END
```

```
SYMBOL TABLE LISTING
-------------------
N A M E        T Y P E   V A L U E    A T T R I B U T E S

ASCBIN. . . . . C ADDR   ----   EXT
BINASC. . . . . C ADDR   ----   EXT
CONSTANT_AREA . C SEG    0056H
DATA_AREA . . . D SEG    000CH
GET_NUM . . . . C ADDR   ----   EXT
NUM1_MSG. . . . C ADDR   001BH  R    SEG=CONSTANT_AREA
NUM1. . . . . . D ADDR   0000H  R    SEG=DATA_AREA
NUM2_MSG. . . . C ADDR   0032H  R    SEG=CONSTANT_AREA
NUM2. . . . . . D ADDR   0004H  R    SEG=DATA_AREA
PUT_CRLF. . . . C ADDR   ----   EXT
PUT_DATA_STR. . C ADDR   ----   EXT
PUT_STRING. . . C ADDR   ----   EXT
SAMPLE. . . . . ----
SCON. . . . . . D ADDR   0098H  A
STACK . . . . . D ADDR   0008H  A
START . . . . . C ADDR   000BH  A

SUM_MSG . . . . C ADDR   004AH  R    SEG=CONSTANT_AREA
SUM . . . . . . D ADDR   0008H  R    SET=DATA_AREA
TH1 . . . . . . D ADDR   008DH  A
TMOD. . . . . . D ADDR   0089H  A
TR1 . . . . . . B ADDR   008BH.6 A
TYPO_MSG. . . . C ADDR   0000H  R    SEG=CONSTANT_AREA

REGISTER BANK(S) USED: 0, TARGET MACHINE(S): 8051

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure 5-1. SAMP1 Listing File (Cont'd.)

Figure 5-1. SAMP1 Listing File (Cont'd.)

```
system-id MCS-51 MACRO ASSEMBLER Vx.y
OBJECT MODULE PLACED IN SAMP2.OBJ
ASSEMBLER INVOKED BY: ASMS: SAMP2.AS1 DEBUG

LOC  OBJ          LINE  SOURCE

                   1           NAME CONSOLE_IO
                   2
                   3     ; IO_ROUTINES segment CODE
                   4           RSEG IO_ROUTINES
                   5     ; This is the console IO routine cluster.
                   6           PUBLIC put_crlf,put_string,put_data_str,get_num
                   7           USING 0
                   8
                   9     ; This routine outputs a Carriage Return and
                  10     ; a Line Feed
                  11     PUT_CRLF:
000D              12     CR EQU 0DH            ; carriage return
000A              13     LF EQU 0AH            ; line feed
                  14     ;
0000 740D         15           MOV A,#cr
0002 120000    F  16           CALL put_char
0005 740A         17           MOV A,#lf
0007 120000    F  18           CALL put_char
000A 22           19           RET
                  20     ;
                  21     ; Routine outputs a null-terminated string located
                  22     ; in CODE memory, whose address is given in DPTR.
                  23     PUT_STRING:
000B E4           24           CLR A
000D 93           25           MOVC A,@A+DPTR
000F 6006         26           JZ exit
0012 120000    F  27           CALL put_char
0012 A3           28           INC DPTR
0013 80F6         29           JMP put_string
0015 22           30     EXIT:
                  31           RET
                  32     ;
                  33     ; Routine outputs a string located in DATA memory,
                  34     ; whose address is in R1 and its length in R2.
                  35     PUT_DATA_STR:
```

Figure 5-2. SAMP2 Listing File

```
0016 E7           36           MOV A,@R1
0017 120000    F  37           CALL put_char
001A 09           38           INC R1
001B DAF9         39           DJNZ R2,put_data_str
001D 22           40           RET
                  41     ;
                  42     ; Routine outputs a single character to console.
                  43     ; The character is given in A.
                  44     PUT_CHAR:
001E 3099FD       45           JNB TI,$
0021 C299         46           CLR TI
0023 F599         47           MOV SBUF,A
0025 22           48           RET
                  49     ;
                  50     ; Get a 4 character string from console
                  51     ; and stores it at the address given in R0.
                  52     ; If RX is received, routine starts over again.
                  53     GET_NUM:
0026 7A04         54           MOV R2,#4         ; set up string length as 4
002B A900         55           MOV R1,AR0        ; R0 value may be needed for restart
                  56     GET_LOOP:
002A 120000    F  57           CALL get_char
                  58     ; Next 4 instr's handle X- the routine starts
                  59     ; over if received
002D C2E7         60           CLR ACC.7         ; clear the parity bit
002F B41805       61           CJNE A,#18H,GO_ON ; if not X- go an
0032 120000       62           CALL put_crlf
0035 80EF         63           JMP get_num
                  64     GO_ON:
0037 F7           65           MOV @R1,A
003B 09           66           INC R1
0039 DAEF         67           DJNZ R2,get_loop
003B 22           68           RET
                  69     ;
                  70     ; Get a single character from console.
                  71     ; The character is returned in A.
                  72     GET_CHAR:
```

Figure 5-2. SAMP2 Listing File (Cont'd.)

```
003C 3098FD        73        JNB   RI,$
003F C29B          74        CLR   RI
0041 E599          75        MOV   A,SBUF
0043 22            76        RET
                   77        ;
                   78        END
```

SYMBOL TABLE LISTING

N A M E	T Y P E	V A L U E	A T T R I B U T E S
ACC.	D ADDR	00E0H	A
ARO.	D ADDR	0000H	A
CONSOLE_IO . .	----	----	
CR	NUMB	000DH	A
EXIT	C ADDR	001SH	R SEG=IO_ROUTINES
GET_CHAR . . .	C ADDR	003CH	R SEG=IO_ROUTINES
GET_LOOP . . .	C ADDR	002AH	R SEG=IO_ROUTINES
GET_NUM. . . .	C ADDR	0026H	R PUB SEG=IO_ROUTINES
GO_DN.	C ADDR	0037H	R SEG=IO_ROUTINES
ID_ROUTINES. .	C SEG	0044H	R REL=UNIT
LF	NUMB	000AH	A
PUT_CHAR . . .	C ADDR	001EH	R SEG=IO_ROUTINES
PUT_CRLF . . .	C ADDR	0000H	R PUB SEG=IO_ROUTINES
PUT_DATA_STR .	C ADDR	0016H	R PUB SEG=IO_ROUTINES
PUT_STRING . .	C ADDR	000BH	R PUB SEG=IO_ROUTINES
RI	B ADDR	0098H.0	A
SBUF	D ADDR	0099H	A
TI	B ADDR	0098H.1	A

REGISTER BANK(S) USED: 0, TARGET MACHINE(S): 8051

ASSEMBLY COMPLETE, NO ERRORS FOUND

Figure 5-2. SAMP2 Listing File (Cont'd.)

```
MCS-51 MACRO ASSEMBLER    NUM_CONVERSION
system-id MCS-51 MACRO ASSEMBLER Vx.y
OBJECT MODULE PLACED IN SAMP3.OBJ
ASSEMBLER INVOKED BY: ASMS1 SAMP3.A51 DEBUG

LOC  OBJ        LINE    SOURCE

                  1             NAME  NUM_CONVERSION
                  2         ;
                  3             NUM_ROUTINES segment CODE
                  4             RSEG  NUM_ROUTINES
                  5         ; This module converts from ASCII to binary
                  6         ; and back. The binary numbers are signed one-byte
                  7         ; integers, i.e. range is -128 to +127. Their
                  8         ; ASCII representation is always 4 char's long-
                  9         ; i.e. a sign followed by 3 digits.
                 10             PUBLIC ascbin, binasc
                 11             USING 0
0030             12     ZERO   EQU   '0'
002B             13     PLUS   EQU   '+'
002D             14     MINUS  EQU   '-'
                 15         ;
                 16         ; This routine converts ASCII to binary.
                 17         ; INPUT- a 4 char string pointed at by R1. The
                 18         ;    number range must be -12B to +127, and the
                 19         ;    string must have 3 digits preceded by a sign.
                 20         ; OUTPUT- a signed one-byte integer, located where
                 21         ;    the input string started (pointed at by R1).
                 22     ASCBIN:
0000 AB01        23         MOV   RO,AR1  ; R1 original value needed later
                 24         ; Compute first digit value, and store it in TEMP
REG              25     TEMP EQU  R3
0002 0B          26         INC   RO
0003 E6          27         MOV   A,@RO
0004 C3          28         CLR   C
002B             29         SUBB  A,#zero
0005 9430        30         MOV   B,#100
0007 7SF064      31         MUL   AB
000B A4          32         MOV   TEMP,A
000B FB          33         ; Compute the second digit value
                 34         INC   RO
```

Figure 5-3. SAMP3 Listing File

```
35   000C 08                   MOV  A,@R0
36   000D E6                   SUBB A,#zero
37   000E 9430                 MOV  8,#10
38   0010 75F00A               MUL  A8
39                           ; Add the value of the second digit to num.
40   0014 28                   ADD  A,TEMP
41   0015 F8                   MOV  TEMP,A
42                           ; Get third digit and its value to total
43   0016 08                   INC  R0
44   0017 E6                   MOV  A,@R0
45   0018 C3                   CLR  C
46   0019 9430                 SUBB A,#zero
47   001B 28                   ADD  A,TEMP
48   001C F8                   MOV  TEMP,A
49                           ; Test the sign and complement the number if the
50                           ; sign is a minus
51   001D E7                   MOV  A,@R1
52   001E 842004               CJNE A,#minus,pos   ; Skip the next 4 instr's
53                                                 ; if the number is positive
54   0021 E8                   MOV  A,TEMP
55   0022 F4                   CPL  A
56   0023 04                   INC  A
57   0024 F8                   MOV  TEMP,A
58                           ; Epilogue- store the result and exit
59                           POS:
60   0025 E8                   MOV  A,TEMP
61   0026 F7                   MOV  @R1,A
62   0027 22                   RET
63
64                           ; This routine converts binary to ASCII.
65                           ; INPUT-a signed 1-byte integer, pointed at by R1
66                           ; OUTPUT- a 4 character string, located where the
67                           ;   input number was (pointed at by R1).
68   00E7                    BINASC:
69                           SIGN bit ACC.7
70                           ; Get the number, find its sign and store its sign
71
```

Figure 5-3. SAMP3 Listing File (Cont'd.)

```
72   0028 E7                   MOV  A,@R1
73   0029 7728                 MOV  @R1,#plus      ; Store a plus sign (over-
74                                                 ; written by minus if needed)
75   002B 30E7D4               JNB  sign,go_on2    ; Test the sign bit
76                           ; Next 3 instructions handle negative numbers
77   002E 7720                 MOV  @R1,#minus     ; Store a minus sign
78   0030 14                   DEC  A
79   0031 F4                   CPL  A
80                           ; Factor out the first digit
81                           GO_ON2:
82   0032 09                   INC  R1             ; store the first digit
83   0033 75F064               MOV  8,#100
84   0036 84                   DIV  A8
85   0037 2430                 ADD  A,#zero
86   0039 F7                   MOV  @R1,A
87                           ; Factor out the second digit
88   003A 09                   INC  R1             ; store the second digit
89   003B E5F0                 MOV  A,B
90   003D 75F00A               MOV  8,#10
91   0040 84                   DIV  A8
92   0041 2430                 ADD  A,#zero
93   0043 F7                   MOV  @R1,A
94                           ; Store the third digit
95   0044 09                   INC  R1             ; store the third digit
96   0045 E5F0                 MOV  A,B
97   0047 2430                 ADD  A,#zero
98   0049 F7                   MOV  @R1,A
99                           ; note that we return without restoring R1
100  004A 22                   RET
101                          ;
102                          END
```

Figure 5-3. SAMP3 Listing File (Cont'd.)

SYMBOL TABLE LISTING

N A M E	T Y P E	V A L U E	A T T R I B U T E S
ACC.	D ADDR	00E0H	A
AR1.	D ADDR	0001H	A
ASCBIN	C ADDR	0000H	R PUB	SEG=NUM_ROUTINES
B.	D ADDR	00F0H	A
BINASC	C ADDR	0028H	R PUB	SEG=NUM_ROUTINES
GO_ON2	C ADDR	0032H	R	SEG=NUM_ROUTINES
MINUS.	NUMB	002DH	A
NUM_CDNVERSION	----	----	REL=UNIT
NUM_RDUTINES	C SEG	004BH	R
PLUS.	C NUMB	002BH	A	SEG=NUM_ROUTINES
POS.	C ADDR	0025H	R
SIGN.	B ADDR	00E0H.7	A
TEMP	REG	R3	
ZERD	NUMB	0030H	A

REGISTER BANK(S) USED: 0, TARGET MACHINE(S): 8051

ASSEMBLY CDMPLETE, NO ERRORS FOUND

Figure 5-3. SAMP3 Listing File (Cont'd.)

---

system-id RLS1

system-id MCS-S1 RELOCATOR AND LINKER, Vx.y, INVOKED BY:
RLS1 SAMP1.OBJ,SAMP2.OBJ,SAMP3.OBJ &
**TO SAMPLE &
**PRINT (SAMPLE.LST) SYMBOLS LINES PUBLICS IXREF

INPUT MODULES INCLUDED
SAMP1.OBJ(SAMPLE)
SAMP2.DBJ(CDNSOLE_ID)
SAMP3.OBJ(NUM_CONVERSION)

LINK MAP FOR SAMPLE(SAMPLE)

TYPE	BASE	LENGTH	RELOCATIDN	SEGMENT NAME
REG	0000H	0008H		"REG BANK 0"
DATA	000BH	000BH	ABSOLUTE	
DATA	0010H	000CH	UNIT	DATA_AREA
CODE	0000H	00SAH	ABSOLUTE	
CODE	00SAH	00S6H	UNIT	CONSTANT_AREA
CODE	00B0H	004BH	UNIT	NUM_ROUTINES
CODE	00FBH	0044H	UNIT	IO_ROUTINES

SYMBOL TABLE FOR SAMPLE(SAMPLE)

VALUE	TYPE	NAME
----	MODULE	SAMPLE
C:00SAH	SEGMENT	CONSTANT_AREA
D:0010H	SEGMENT	DATA_AREA
C:0007SH	SYMBOL	NUM1_MSG
D:0010H	SYMBOL	NUM1
C:000BCH	SYMBOL	NUM2_MSG
D:0014H	SYMBOL	NUM2
0:009BH	SYMBOL	SCON

mm/dd/yy PAGE 1

Figure 5-4. RL51 Output File

```
D:0008H     SYMBOL    STACK
C:000BH     SYMBOL    START
C:00A4H     SYMBOL    SUM_MSG
0:0018H     SYMBOL    SUM
0:00BDH     SYMBOL    TH1
0:00B9H     SYMBOL    TMO0
B:00B8H.6   SYMBOL    TR1
C:00SAH     SYMBOL    TYPO_MSG
-------     ENDMOD    SAMPLE
D:0E0H      MODULE    CONSOLE_IO
D:0000H     SYMBOL    ACC
H:000DH     SYMBOL    AR0
C:0110H     SYMBOL    CR
C:0137H     SYMBOL    EXIT
C:0125H     SYMBOL    GET_CHAR
C:0121H     SYMBOL    GET_LOOP
C:0132H     PUBLIC    GET_NUM
C:00FBH     SEGMENT   GO_ON
H:000AH     SYMBOL    IO_ROUTINES
C:0119H     SYMBOL    LF
                      PUT_CHAR

system-id RLS1

C:00FBH     PUBLIC    PUT_CRLF
C:0111H     PUBLIC    PUT_DATA_STR
C:0106H     PUBLIC    PUT_STRING
B:0098H     SYMBOL    RI
0:0099H     SYMBOL    SBUF
B:009BH.1   SYMBOL    TI
-------     ENDMOD    CONSOLE_IO

-------     MODULE    NUM_CONVERSION
0:00EQH     SYMBOL    ACC
0:0001H     SYMBOL    AR1
C:00B0H     PUBLIC    ASCBIN
0:00F0H     SYMBOL    B
C:000BH     PUBLIC    BINASC
```

mm/dd/yy   PAGE   2

Figure 5-4. RL51 Output File (Cont'd.)

```
C:00E2H     SYMBOL    GO_ON2
H:0020H     SYMBOL    MINUS
C:00B0H     SEGMENT   NUM_ROUTINES
C:000SH     SYMBOL    PLUS
C:000SH     SYMBOL    POS
B:00E0H.7   SYMBOL    SIGN
H:0030H     SYMBOL    ZERO
-------     ENDMOD    NUM_CONVERSION


INTER-MODULE CROSS-REFERENCE LISTING
------------------------------------

NAME . . . . .   USAGE   MODULE NAMES
-----------      -----   ------------
ASCBIN. . . .    CODE:   NUM_CONVERSION SAMPLE
BINASC. . . .    CODE:   NUM_CONVERSION SAMPLE
GET_NUM . . .    CODE:   CONSOLE_IO SAMPLE
PUT_CRLF. . .    CODE:   CONSOLE_IO SAMPLE
PUT_DATA_STR.    CODE:   CONSOLE_IO SAMPLE
PUT_STRING. .    CODE:   CONSOLE_IO SAMPLE
```

Figure 5-4. RL51 Output File (Cont'd.)

Using the Locating Controls

The second example shows how to use the PRECEDE control to specify an order for data segments, in this case because the RL51 algorithm for locating segments results in a segment being left out.

The program is named TEST01. After assembly, the listing of TEST01.OBJ is as shown in figure 5-5. The program's code sequence is irrelevant to the example. The two DATA segments, SEG1 and SEG2, and the BIT segment, BIT3, are the points of interest for this example.

SEG1 is 21H bytes long; SEG2, 50H bytes long; SEG3, one bit long. The assembler listing also shows working.register bank 0 (8 bytes long, absolutely located at addresses 00H through 07H).

All these segments are to be located in the on-chip data RAM of an 8051. For the 8051, the directly-addressable on-chip data RAM is 80H bytes long (addresses 00H through 7FH); addresses 20H through 2FH are bit-addressable. The working registers may occupy the first 20H bytes of the space. To see what RL51 does with this program, enter the command

```
RLS1 TEST01.OBJ
```

The RL51 listing file is shown in figure 5-6. ERROR 107 informs us that the locate attempt for SEG1 would overflow the data space; SEG1 was ignored (not located) for this reason. The link map shows the following assignments for the remaining segments:

Addresses	Segment
00H - 07H	Register Bank 0
08H - 1FH	GAP
20H	SEG3 (one bit at bit location 0)
20H.1 - 20H.7	GAP
21H - 71H	SEG2 (50H bytes)

After these segments have been located, there is not enough room for SEG1 (21H bytes). However, there would be enough room if SEG1 were located before the BIT segment. To obtain this result, the command is

```
RLS1 TEST01.OBJ PRECEDE(SEG1)
```

The RL51 listing file for this example is shown in figure 5-7. The PRECEDE control caused the link mapping to be as follows:

Addresses	Segment
00H - 07H	Register Bank 0
08H - 28H	SEG1 (21H bytes)
29H	SEG3 (one bit at bit location 0)
29H.1 - 29H.7	GAP
2AH - 7AH	SEG2 (50H bytes)

Refer to Chapter 2 for details on RL51's allocating algorithm.

Using RL51 with PL/M-51 Modules

The third example shows how to use RL51 with object modules produced by PL/M-51. The example shows the use of PLM51.LIB and demonstrates PL/M-51 generated segments and the PL/M-51 to ASM51 linkage.

The entire application introduces a way to halt ICE-51 the 8051 In-Circuit Emulator Program, at run time. The procedure CHECK_EQUAL in the PL/M-51 module CHK_EQ checks if an arithmetic expression is true. If yes, it calls the HALT_ICE assembler routine, which causes ICE51 to stop the program that is currently running. The code of the program is irrelevant; the example merely intends to show the program development process.

The PLM51 main module CHK_EQ is compiled by

```
PLMS1 CHKEQ.PS1 DEBUG PW(90)
```

The output of the compilation is shown in figure 5-8.

The ASM51 module HLTICE is assembled by

```
ASMS1 HLTICE.AS1 DEBUG PW(90)
```

The output of the compilation is shown in figure 5-9.

RL51 is invoked by the following command:

```
RLS1 CHKEQ.OBJ,HLTICE.OBJ,PLMS1.LIB IXREF PW(72)
```

RL51 links the two pre-translated input modules, along with the mandatory library PLM51.LIB. PLM51.LIB must be linked whenever a PL/M-51 module participates in the linkage. The listing files are shown in figures 5-8 and 5-9. The result of the linkage is shown in figure 5-10.

The result of a linkage process that includes PL/M-51 modules deserves an explanation. The following paragraphs describe the modules, segments, and symbols that appear in the output listing of such a linkage. The explanation refers to the actual example (figure 5-10).

In addition to the two input modules CHK_EQ and HALT_ICE, RL51 pulled some modules from PLM51.LIB. The two modules ?P0034 and ?P0038 contain common PL/M-51 run-time routines and were pulled to resolve calls to those routines in the CHK_EQ module. The module ?P1V0R contains the initialization routine (set the stack pointer, set PSW), and is pulled whenever a linkage process encounters a main module written in PL/M-51.

The segments BYTES, BITS, and PROG are the user segments as defined in the ASM51 HALT_ICE module. The code segments ?P0034S, ?P0038S and ?P1V0RS are the code segments of the previously explained run-time routines.

All segments whose names are of the form ?CHK_EQ?any are segments generated by PL/M-51 as result of compiling module CHK_EQ. The prefix ?CHK_EQ? indicates that the segment belongs to the CHK_EQ module. The suffix indicates the segment type; e.g, PR stands for the PRogram CODE segment, CO for the COnstant CODE segment, DT for DATA segment, and BI for BIT segment.

On-chip segment names may be followed by a register bank number (0-3). This number indicates the register bank that must be in effect while data in this segment is accessed.

The ?STACK segment was discussed before. Note that this segment is not supplied by the user, but is pulled automatically from PLM51.LIB because the main module is written in PL/M-51. The absolute segment at 0000H-0002H contains the reset vector, which consists of a JUMP to the initialization routine contained in the ?P1V0RS segment.

Most of PL/M-51-generated relocatable segments have the UNIT relocation type. A frequent exception is the program code segment (?CHK_EQ?PR), which is INBLOCK whenever a module is compiled under ROM (MEDIUM), which is the default used by the compiler. Another (less frequent) exception is the BITAD-DRESSABLE DATA segment generated when bit structures are declared within the PL/M-51 source program.

User symbols appear in the symbol table and the IXREF report. Symbols whose names are equal to segments and modules defined previously represent entry points in the appropriate modules/segments pulled from PLM51.LIB (e.g., the symbol ?P0034 is a code address in the module ?P0034).

Symbols in the format *?procedure*?BYTE or *?procedure*?BIT (e.g., ?HALT_ICE?BYTE) are DATA and BIT addresses used for passing parameters to the appropriate external procedures (as implied by the name). BYTE and WORD parameters are placed at DATA address starting at, for example, ?HALT_ICE?BYTE. BIT parameters are placed at BIT address starting at ?HALT_ICE?BIT (see also the *PL/M-51 User-s Guide* about PL/M-51 linkage to ASM51).

```
MCS-51 MACRO ASSEMBLER    TEST01
system-id MCS-51 MACRO ASSEMBLER Vx.y
OBJECT MODULE PLACED IN TEST01.OBJ
ASSEMBLER INVOKED BY: ASM51 TEST01.SRC PRINT (:CO:) &
**OBJECT (TEST01.OBJ)

LOC  OBJ        LINE   SOURCE

                  1    ; This test shows the use of the
                  2    ; PRECEDE locating control.
                  3    ; One bit causes failure of the
                  4    ; RLS1 allocation algorithm,
                  5    ; but the PRECEDE control fixes it.
                  6    ;
                  7           NAME   test01
                  8    ;
                  9    prog   SEGMENT   CODE
                 10    seg1   SEGMENT   DATA
                 11    seg2   SEGMENT   DATA
                 12    seg3   SEGMENT   BIT
                 13    ;
                 14           RSEG   prog
                 15    ;
                 16    ; Code segment.
                 17    ;
0000 434F4445   18           DB     'CODE IS IRRELEVANT'
     20495320
     49525245
     4C455641
     4E54
                 19
0000             20           RSEG   seg1
                 21           DS     021H
0000             22           RSEG   seg2
                 23           DS     050H
0000             24           RSEG   seg3
                 25           DBIT   001H
                 26
                 27
                 28
                 29           END
```

Figure 5-5. TEST01 Assembly Listing File

system-id RLS1

system-id MCS-51 RELOCATOR AND LINKER, Vx.y, INVOKED BY:
RLS1 TEST01.OBJ

INPUT MODULES INCLUDED
TEST01.OBJ(TEST01)

LINK MAP FOR TEST01(TEST01)

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME
REG	0000H	0008H		"REG BANK 0"
	0008H	0018H		... GAP ...
BIT	0020H	0000H.1	UNIT	SEG3
	0020H.1	0000H.7	UNIT	... GAP ...
DATA	0021H	00S0H	UNIT	SEG2
CODE	0000H	0012H	UNIT	PROG

IGNORED SEGMENTS
SEG1

- ERROR 107: ADDRESS SPACE OVERFLOW
  SPACE: DATA
  SEGMENT: SEG1

Figure 5-6. RL51 Listing File without PRECEDE

system-id RLS1

system-id MCS-51 RELOCATOR AND LINKER, Vx.y, INVOKED BY:
RLS1 TEST01.OBJ PRECEDE(SEG1)

INPUT MODULES INCLUDED
TEST01.OBJ(TEST01)

LINK MAP FOR TEST01(TEST01)

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME
REG	0000H	000BH		"REG BANK 0"
DATA	0008H	0021H	UNIT	SEG1
BIT	0029H.1	0000H.7	UNIT	SEG3
				... GAP ...
DATA	002AH	00S0H	UNIT	SEG2
CODE	0000H	0012H	UNIT	PROG

Figure 5-7. RL51 Listing File with PRECEDE

```
PL/M-51 COMPILER       -- ICES1 - Check/Halt --          mm/dd/yy    PAGE    1

system-id PL/M-51 Vx.y
COMPILER INVOKED BY: plm51 chteq.p51 debug pw(90)

              $title ('-- ICES1 - Check/Halt --')
              /*.........................................*/
              /*  Check_equal:                           */
              /*      Check if comparison yields the     */
              /*      expected result. If not, call      */
              /*      assembler routine to return to ICE.*/
              /*.........................................*/

     1    1   chk_eq:
               DO;

     2    1   halt_ice:
               PROCEDURE (val1, val2, eq_switch) EXTERNAL;
     3    2     DECLARE (val1, val2) WORD, eq_switch BIT;  END;

     5    1   check_equal:
               PROCEDURE (val1, val2, eq_switch) PUBLIC;
     6    2     DECLARE
                  (val1, val2) WORD,
                   eq_switch    BIT;

     7    2     IF ((val1 <> val2) <> eq_switch)
                  THEN CALL halt_ice (val1, val2, eq_switch);
     9    1   END check_equal;

               /* dummy main program */

    10    1   DECLARE
                  p1 WORD CONSTANT (3),
                  s1 WORD;

    11    1     CALL check_equal(p1*s1, 27//s1, 1);

    12    1   END chk_eq;
```

Figure 5-8. PL/M-51 Listing File of CHK_EQ

```
MODULE INFORMATION:                (STATIC+OVERLAYABLE)
    CODE SIZE                = 00S7H     B7D
    CONSTANT SIZE            = 0002H      2D
    DIRECT VARIABLE SIZE     = 02H+04H    2D+    4D
    INDIRECT VARIABLE SIZE   = 00H+00H    0D+    0D
    BIT SIZE                 = 00H+01H    0D+    1D
    BIT-ADDRESSABLE SIZE     = 00H+00H    0D+    0D
    AUXILIARY VARIABLE SIZE  = 0000H      0D
    MAXIMUM STACK SIZE       = 0006H      6D
    REGISTER-BANK(S) USED:     0
    34 LINES READ
    0 PROGRAM ERROR(S)
END OF PL/M-51 COMPILATION
```

Figure 5-8. PL/M-51 Listing File of CHK_EQ (Cont'd.)

```
MCS-51 MACRO ASSEMBLER     '.. ICES1 - Halt ..'        mm/dd/yy   PAGE   1

system-id MCS-51 MACRO ASSEMBLER Vx.y
OBJECT MODULE PLACED IN HLTICE.OBJ
ASSEMBLER INVOKED BY: asm51 hltice.as1 debug pw(90)

LDC  OBJ        LINE    SOURCE
                 1      $title ('.. ICES1 - Halt ..')
                 2      ;*************************************
                 3      ;*  Halt_ice:                       *
                 4      ;*     Store word parameters in R4S, R67,  *
                 5      ;*     Bit in C and execute AS instruction *
                 6      ;*     to return to ICE.            *
                 7      ;*                                  *
                 8      ;*************************************
                 9              NAME    halt_ice
                10              PUBLIC  halt_ice, ?halt_ice?bit, ?halt_ice?byte
                11      bits    SEGMENT BIT
                12      bytes   SEGMENT DATA
                13      prog    SEGMENT CODE
                14      ;
                15              RSEG    bits
                16      ?halt_ice?bit:
                17      bit_par:        DBIT 1
                18
                19      ;
                20              RSEG    bytes
                21      ?halt_ice?byte:
                22      first_par:      DS 2
 0000           23      second_par:     DS 2
 0000           25
 0002           26      ;
                27              RSEG    prog
                28      halt_ice:
 0000 AC00  F   29              MOV     R4,first_par     ; move 1st par to place
 0002 AD00  F   30              MOV     R5,first_par+1   ; move 2nd par to place
 0004 AE00  F   31      MOV     R6,second_par
```

Figure 5-9. ASM51 Listing File of HLTICE

```
 0006 AF00  F   32              MOV     R7,second_par+1
 000B A200  F   33              MOV     C,bit_par        ; move bit par to place
 000A A5     34      ;
 000B 00     36              DB      0ASH             ; illegal op-code.
 000C 00     37              NOP                      ; Will stop ICE-51 if you
 000D 00     38              NOP                      ; type "GO TIL OPC IS AS"
 000E 22     39              NOP
                40              RET                      ; you can continue after
                41              END                      ; stop.

MCS-51 MACRO ASSEMBLER     '.. ICES1 - Halt ..'

SYMBOL TABLE LISTING
------ ----- -------

N A M E           T Y P E   V A L U E     A T T R I B U T E S

?HALT_ICE?BIT.    B ADDR    0000H.0 R PUB    SEG=BITS
?HALT_ICE?BYTE    D ADDR    0000H.0 R PUB    SEG=BYTES
BIT_PAR. . . .    B ADDR    0000H.0 R        SEG=BITS
BITS . . . . .    B SEG     0001H   R        REL=UNIT
BYTES. . . . .    D SEG     0004H   R        REL=UNIT
FIRST_PAR. . .    D ADDR    0000H   R        SEG=BYTES
HALT_ICE . . .    C ADDR    0000H   R PUB    SEG=PROG
PROG . . . . .    C SEG     000FH   R        REL=UNIT
SECOND_PAR . .    D ADDR    0002H   R        SEG=BYTES

REGISTER BANK(S) USED: 0, TARGET MACHINE(S): 8051

ASSEMBLY COMPLETE, NO ERRORS FOUND
```

Figure 5-9. ASM51 Listing File of HLTICE (Cont'd.)

system-Id RLS1   mm/dd/yy   PAGE 1

ISIS-II MCS-51 RELOCATOR AND LINKER, Vx.y, INVOKED BY:
RLS1 CHKEO.OBJ,HLTICE.OBJ,PLMS1.LIB 1XREF PW(72)

INPUT MODULES INCLUDED
  CHKEO.OBJ(CNK_EO)
  HLTICE.OBJ(HALT_ICE)
  PLMS1.LIB(?P0034)
  PLMS1.LIB(?P0038)
  PLMS1.LIB(?PIVOR)

LINK MAP FOR CHKEO(CNK_EO)

TYPE	BASE	LENGTH	RELOCATION	SEGMENT NAME
REG	0000H	000BH		"REG BANK 0" BYTES
DATA	0008H	0004H	UNIT	?CNK_EO?DT?0
DATA	000CH	0004H	UNIT	?CNK_EO?DT
DATA	0010H	0002H	UNIT	*** GAP ***
	0012H	000EH		
BIT	0020H	0000H.1	UNIT	?CNK_EO?BI?0
BIT	0020H.1	0000H.6		BITS
	0020H.2			*** GAP ***
IDATA	0021H	0001H	UNIT	?STACK
CODE	0000H	0003H	ABSOLUTE	?CNK_EO?PR
CODE	0003H	00S7H	INBLOCK	?P0038S
CODE	00SAH	0016H	UNIT	PROG
CODE	0070H	000FH	UNIT	?P0034S
CODE	007FH	000CH	UNIT	?PIVORS
CODE	008BH	0009H	UNIT	?CNK_EO?CD
CODE	0094H	0002H	UNIT	

Figure 5-10. RL51 Listing File of CHK EQ

---

system-Id RLS1   mm/dd/yy   PAGE 2

SYMBOL TABLE FOR CHKEO(CNK_EO)

VALUE	TYPE	NAME
-------	MODULE	CNK_EO
C:0003H	PUBLIC	CHECK_EQUAL
C:0033H	SYMBOL	CHK_EO
	PROC	CHECK_EQUAL
D:000CH	SYMBOL	VAL1
D:000EH	SYMBOL	VAL2
B:0020H	SYMBOL	EO_SWITCH
-------	ENDPROC	CHECK_EQUAL
C:0094H	SYMBOL	PI
D:0010N	SYMBOL	SI
C:0033H	LINE#	1
C:0003H	LINE#	S
C:0001FH	LINE#	7
C:0032H	LINE#	B
	LINE#	9

system-Id RLS1

VALUE	TYPE	NAME
C:0033H	LINE#	11
C:00SAH	LINE#	12
-------	ENDMOD	CHK_EO
-------	MODULE	HALT_ICE
B:0020N.1	PUBLIC	?HALT_ICE?BIT
D:0008H	PUBLIC	?HALT_ICE?BYTE
B:0020H.1	SYMBOL	BIT_PAR
D:0008N	SEGMENT	BITS
D:0008H	SEGMENT	BYTES
C:0070H	SYMBOL	FIRST_PAR
C:0070H	PUBLIC	HALT_ICE
C:000AH	SEGMENT	PROG
D:000AH	SYMBOL	SECOND_PAR
	ENOMOD	HALT_ICE

Figure 5-10. RL51 Listing File of CHK EQ (Cont'd.)

```
system-id RLS1                                    mm/dd/yy   PAGE 3

INTER-MODULE CROSS-REFERENCE LISTING

NAME . . . . . . . . . . .   USAGE        MODULE NAMES
. . . . . . . . . . . . . .   . . . . .    . . . . . . . .

?CHECK_EQUAL?BIT . . . . .   BIT;         CHK_EQ
?CHECK_EQUAL?BYTE. . . . .   DATA;        CHK_EQ
?HALT_ICE?BIT. . . . . . .   BIT;         HALT_ICE   CHK_EQ
?HALT_ICE?BYTE . . . . . .   DATA;        HALT_ICE   CHK_EQ
?P0034 . . . . . . . . . .   CODE;        ?P0034     CHK_EQ
?P0038 . . . . . . . . . .   CODE;        ?P0038     CHK_EQ
?PINOR . . . . . . . . . .   CODE;        CHK_EQ     ?PIVOR
?PIVOR . . . . . . . . . .   CODE;        ?PIVOR     CHK_EQ
?PSWOR . . . . . . . . . .   NUMB;        CHK_EQ     ?PIVOR
CHECK_EQUAL. . . . . . . .   CODE;        CHK_EQ
HALT_ICE . . . . . . . . .   CODE;        HALT_ICE   CHK_EQ
```

Figure 5-10. RL51 Listing File of CHK_EQ (Cont'd.)

LIB51 Librarian 6

Introduction

LIB51 allows you to create, modify, and examine library files. It may be executed in interactive or noninteractive mode. In both cases, LIB51 can be invoked directly or by a submit file.

Invoking LIB51

Noninteractive Mode

Following is the general syntax for invoking in non-interactive mode:

[*directory* | *device*] L I B 5 1   *command*

The librarian will then respond with the sign-on message. It then executes the given command and returns immediately to the host operating system.

Interactive Mode

Following is the general syntax for invoking LIB51 in interactive mode:

[*directory* | *device*] L I B 5 1

LIB51 will then respond with its sign-on message. It will then present the prompt (*), requesting that you enter LIB51 commands. After each command is executed, another prompt will appear as the librarian awaits entry of the next command. This process continues until the EXIT command is entered, thus terminating LIB51.

Character Set

The LIB51 character set consists of the letters A–Z, the digits 0–9 and the special characters ?, @, and _

LIB51 Commands

Table 6-1 summarizes the LIB51 commands.

Command Entry

It is often necessary to extend the LIB command to more than one line. Use an ampersand (&) to indicate that you have not entered the complete command and are extending it to another line. The ampersand continuation character may be placed anywhere that a space would normally appear in the command line. That is, the continuation character may be placed before or after commas or parentheses and before or after control words. Any characters that appear on a line to the right of the ampersand and to the left of the carriage return terminating the line are ignored.

Table 6-1. LIB51 Commands

Command	Abbreviation	Description
ADD { file[(module[,...])] } [,...] TO library_file	A	Adds modules to a library
CREATE library file	C	Creates a library file
DELETE library_file(module[,...])	D	Deletes modules from a library
EXIT	E	Terminates session with LIB51
EXTRACT { file[(module[,...])] } [,...] TO file	X	Extracts modules from libraries
HELP	H	Displays syntax of LIB51 commands
LIST { file[(module[,...])] } [,...] [TO file] [PUBLICS]	L [P]	Lists modules contained in librar- ies, and optionally lists all publics
REPLACE { file[(module[,...])] } [,...] IN library_file	R	Replaces modules in a library

Whenever you enter the continuation character, LIB51 responds by beginning a new line with the continuation prompt—two asterisks (**). LIB51 then waits for you to enter the additional line of input. If you cannot complete the command on the second line, use more ampersands to continue the process until the command is completely entered.

A semicolon may be placed on any command to start a comment. LIB51 will ignore all characters that appear to the right of the semicolon and to the left of the carriage return that terminates the line. If you enter an ampersand to the right of a semicolon, the ampersand will be treated as part of a comment and not as the continuation character.

Command Descriptions

Following are the descriptions of the LIB51 commands.

ADD

Syntax

```
ADD {file[(module[,...] ) ] } [,...] TO library_file
```

Abbreviation

A

Description

The ADD command allows you to add the specified files to the library file specified as the destination.

The input filenames may be the names of ordinary object files or object library files. If the input file is an ordinary object file, all modules contained within that file will be added to the designated library. The ordinary object file may have been produced by a translator/assembler, RL51, or the EXTRACT command of LIB51.

If the input file is a library file, it may be specified with or without a list of module names. If you do not specify a list of module names, all of the modules contained in the input library will be added to the destination library. If you do specify the list of module names, only those modules specified in the command are added to the destination library.

The destination library must already exist before the ADD command is entered.

Examples

1. `ADD SIN,COS,TAN TO USER.LIB`

   This command adds the three files SIN, COS, and TAN to the destination library USER.LIB.

2. `ADD LIB.TMP(MOD1,MOD2,MOD3) TO &`
   `**PROJ.TOM`

   This command adds the three modules MOD1, MOD2, and MOD3 of the library LIB.TMP to the destination library PROJ.TOM. Note the use of the ampersand to continue the command.

CREATE

Syntax

CREATE *library file*

Abbreviation

C

Description

The CREATE command causes an empty library file that has been named in the command to be created. If the file already exists, an error message is issued and the command terminates.

Examples

1. •CREATE SLEAZO.LIB

   This command creates the empty library file SLEAZO.LIB.

DELETE

Syntax

DELETE, *library file(module name* [, . . .])

Abbreviation

D

Description

The DELETE command removes the specified modules from the designated library file. Modules can be deleted from only one library at a time. If any of the elements specified for deletion cannot be located, a warning is issued.

Examples

1. •DELETE SLEAZO.LIB(TRUTH,VALUE)

   This command deletes the modules TRUTH and VALUE from the library SLEAZO.LIB.

EXIT

Syntax

E X I T

Abbreviation

E

Description

In interactive mode, the EXIT command causes LIB51 to terminate—thereby causing control to be returned to the operating system. In noninteractive mode, the EXIT command is ignored.

Examples

1. `*EXIT`

EXTRACT

Syntax

E X T R A C T {file[(module[,...])]} [,...] TO file

Abbreviation

X

Description

The EXTRACT command builds an ordinary object file from the specified files and library members. The extracted files are not deleted; they remain unchanged (i.e., they are nondestructively copied to their destination).

Examples

1. `*EXTRACT SLEAZO.LIB(WORTH,FREE) TO PROLES.OBJ`

   The modules WORTH and FREE are nondestructively extracted from SLEAZO.LIB and placed in PROLES.OBJ.

HELP

Syntax

HELP

Abbreviation

H

Description

The HELP command causes a summary of the syntax of the LIB51 command to appear on the console. Use the HELP command to obtain this information about LIB51 if you require help when entering commands. The following information will appear on the screen:

```
ADD       {file[ (module[ ,...])]} [ ,...] TO  library_file
CREATE    library file
DELETE    library_file(module[ ,...])
EXIT
EXTRACT   {file[ (module[ ,...])]} [ ,...] TO  file
HELP
LIST      {file[ (module[ ,...])]} [ ,...] [ TO  file] [PUBLICS]
REPLACE   {file[ (module[ ,...] )]} [ ,...] IN  library_file
```

LIST

Syntax

LIST {file[(module[,...])]} [,...] [TO file] [PUBLICS]

Abbreviation

L [P]

Description

The LIST command prints the names of the modules, and, optionally, the names of the public symbols (if you specify PUBLICS) to the specified destination output file. If you do not enter the TO clause, the listing will be directed to the console output. PUBLICS specifies that in addition to the module names, all public symbols contained in those modules will be listed.

Examples

1. `*LIST USER.LIB`

   The names of all modules in the library USER.LIB are listed.

2. `*LIST USER.LIB(TEMP) PUBLICS`

   All public symbols in the module TEMP in the library USER.LIB are listed.

REPLACE

Syntax

REPLACE {file[(module[,...])]} [,...] IN library_file

Abbreviation

R

Description

The REPLACE command allows you to replace object modules in the designated library file with a new version. If a module designated to be replaced does not already exist in the library in an older version, the newer version is simply added to the library.

Examples

I. `REPLACE WORTH,FREE IN SLEAZO.LIB`

The newer version of WORTH is added to the library SLEAZO.LIB; the new file FREE is also added.

Summary of RL51 Controls A

Table of Basic Definitions

Table A-1 gives definitions of basic terms used in the command format summary.

Table A-1. Definitions of Common Terms

Term	Definition
name	Names can be from 1 to 40 characters long and must be composed of letters A - Z, digits 0 - 9, or special characters (?. @. _). The first character must be a letter or a special character.
module-name	Same as name.
segment-name	Same as name.
pathname	A valid filename reference or device reference. See next two items for examples.
filename	A reference to a disk file.
device	A reference to a non-disk device. Examples: :LP:, :CO:, :TO:
value	A 16-bit unsigned integer. Examples: 1011B. 3040, 4096D (or just 4096), 0C300H
address	Same as value.

RL51 Command Format Summary

Here is a summary of the syntax of the RL51 invocation command. Refer to the Preface for an explanation of the command format notation.

The RL51 command has the overall format

[directory|device] RL51 input-list [TO output-file] [control-list]

where

directory|device := ; the directory or device where RL51 resides.

input-list := input-file [module-list] [,...]

input-file := filename ; see table A-1

module-list := (module-name [,...])

module-name := ; see table A-1

output-file := filename ; see table A-1

control-list := control ...

control : = { listing-control
              linking-control
              locating-control
              configuration-control
              overlay-control }

listing-control : = { print
                       pagewidth
                       map
                       symbols
                       publics
                       lines
                       ixref }

print : = { PRINT [(pathname)]
            NOPRINT }

pathname : = ; see table A-1

pagewidth : = PAGEWIDTH (value)

value : = see table A-1

map : = { MAP
          NOMAP }

symbols : = { SYMBOLS
              NOSYMBOLS }

publics : = { PUBLICS
              NOPUBLICS }

lines : = { LINES
            NOLINES }

ixref : = { IXREF [selection-list]
            NOIXREF }

selection-list : = (selection-item [, ...])

selection-item : = { generated
                     libraries }

generated : = { GENERATED
                NOGENERATED }

libraries : = { LIBRARIES
                NOLIBRARIES }

linking-control : = { NAME (module-name)
                      debugsymbols
                      debuglines
                      debugpublics }

debugsymbols : = { DEBUGSYMBOLS
                   NODEBUGSYMBOLS }

debuglines : = { DEBUGLINES
                 NODEBUGLINES }

debugpublics : = { DEBUGPUBLICS
                   NODEBUGPUBLICS }

locating-controls : = { PRECEDE
                        DATA
                        BIT
                        IDATA
                        STACK
                        XDATA
                        CODE } (segment [, ...])

segment : = segment-name [(address)]

segment-name : = ; see table A-1
address : = ; see table A-1

configuration-control : = ramsize

ramsize : = RAMSIZE (value)

value : = ; see table A-1

overlay-control : = { OVERLAY
                      NOOVERLAY [((overlay-unit[, ...]))] }

overlay-unit : = ov-module-name calls ov-module-name

ov-module-name : = { *
                     module-name }

module-name : = ; see table A-1

calls : = > or]

Tables of Listing, Linking, Locating, and OverlayingControls

Tables A-2 through A-6 describe the RL51 controls. Table A-7 gives abbreviations for the controls.

Notes On Locating Controls

The following notes refer to table A-4.

1. Bit addresses for non-BIT segments in the BIT control must be on byte boundaries; that is, they must be divisible by eight. (BIT-type segments can be aligned on bit boundaries.)

2. The range of addresses for the IDATA control is dependent on the target machine. The 8051 has 128 bytes (addresses 00H — 7FH). See the RAMSIZE control in this context.

3. The STACK control specifies which segments are to be allocated uppermost in the IDATA space. The memory accessed starts after the highest on-chip RAM address occupied by any previously allocated segment, and continues to the top of the IDATA space.

NOTE

This control has no other effect on any segments.

The IDATA ?STACK segment, if it exists, is placed higher than segments that were mentioned in the STACK control.

Table A-2. Listing Controls and Switches

Control	Effect
PRINT [(pathname)]	Sends the listing file to the file or device specified by pathname.
NOPRINT	Suppresses the listing file; overrides any of the following listing controls.
PAGEWIDTH (value)	Specifies the maximum page width to be used.
MAP	Outputs memory map to link summary.
NOMAP	Suppresses memory map.
SYMBOLS	Outputs local symbols to symbol table.
NOSYMBOLS	Suppresses local symbols.

Table A-2. Listing Controls and Switches (Cont'd.)

Control	Effect
PUBLICS	Outputs public symbols to symbol table.
NDPUBLICS	Suppresses public symbols.
LINES	Outputs line numbers to symbol table (high-level language translators only).
NOLINES	Suppresses line numbers.
IXREF [(selection-list)]	Appends intermodule cross-reference report to print file.
NOIXREF	Suppresses the intermodule cross-reference report.

NOTE: The default for any control (except IXREF) is the positive form (PRINT, MAP, SYMBOLS, PUBLICS, and LINES).

Table A-3. Linking Controls and Switches

Control	Effect
NAME (module-name)	Specifies the name of the output module. If the NAME control is omitted, the output module name defaults to the name of the first input module processed.
DEBUGSYMBOLS	Copies local symbol information to output file.
NODEBUGSYMBOLS	Suppresses local symbols.
DEBUGPUBLICS	Copies public symbol information to output file.
NODEBUGPUBLICS	Suppresses public symbols.
DEBUGLINES	Copies line number information (high-level language translators only) to output file.
NODEBUGLINES	Suppresses line numbers.

NOTE: For all linking controls except NAME, the default is the positive form (DEBUGSYMBOLS, DEBUGPUBLICS, and DEBUGLINES).

Table A-4. Locating Controls

Control	Address Space	Address Range (Hex)	Segment Types (and Attributes)
PRECEDE	Register banks and bit-addressable space in on-chip data RAM	00H-2FH	DATA (UNIT-aligned); IDATA
BIT	Bit-addressable space in on-chip data RAM	00H - 7FH (see note 1)	BIT; DATA; IDATA
DATA	Directly-addressable on-chip data RAM	00H - 7FH	DATA (UNIT-aligned); IDATA
IDATA	Indirectly-addressable on-chip data RAM	00H - 0FFH (see note 2)	IDATA
STACK	Same as IDATA (see note 3)	Same as IDATA	Same as IDATA
XDATA	External data RAM	0 - 0FFFFH	XDATA
CODE	Code memory	0 - 0FFFFH	CODE

Table A-5. Configuration Control

Control	Effect
RAMSIZE (value)	Specifies the amount of on-chip RAM the object is aimed to.

Table A-6. Overlay Controls

Control	Effect
OVERLAY (overlay-units)	Overlays data segments, based on the information in the module declarations and in the overlay units.
NOOVERLAY	Suppresses the overlaying of data segments.

Table A-7. Abbreviations for Command Words

Command Word	Abbreviation
BIT	BI
CODE	CO
DATA	DT
DEBUGLINES	DL
DEBUGPUBLICS	DP
DEBUGSYMBOLS	DS
GENERATED	GN
IDATA	ID
IXREF	IX
LIBRARIES	LB
LINES	LI
MAP	MA
NAME	NA
NODEBUGLINES	NODL
NODEBUGPUBLICS	NODP
NODEBUGSYMBOLS	NODS
NOGENERATED	NOGN
NOIXREF	NOIX
NOLIBRARIES	NOLB
NOLINES	NOLI
NOMAP	NOMA
NOOVERLAY	NOOL
NOPRINT	NOPR
NOPUBLICS	NOPL
NOSYMBOLS	NOSB
OVERLAY	OL
PAGEWIDTH	PW
PRECEDE	PC
PRINT	PR
PUBLICS	PL
RAMSIZE	RS
STACK	ST
SYMBOLS	SB
TO	TO
XDATA	XD

RL51 error messages describe warnings, errors, and fatal errors. A warning is a detected condition that may or may not be what the user desired; a warning does not terminate the link/locate operation. An error does not terminate operation, but probably results in an output module that cannot be used. A fatal error terminates operation of RL51.

This appendix lists the warning, error, and fatal error messages in that order. The text of each message is in UPPER CASE. A brief explanation of the probable cause for the error condition accompanies each error message.

Warnings

```
WARNING 1: UNRESOLVED EXTERNAL SYMBOL
SYMBOL:     external-neme
MODULE:     file-name(module-name)
```

The specified external symbol, requested in the specified module, has no matching public symbol in any of the input modules.

```
WARNING 2: REFERENCE MADE TO UNRESOLVED EXTERNAL
SYMBOL:     external-name
MODULE:     file-name(module-name)
REFERENCE:  code-address
```

The specified unresolved external is referenced in the specified module at the specified code address.

```
WARNING 3: ASSIGNED ADDRESS NOT COMPATIBLE WITH
           ALIGNMENT
SEGMENT:    segment-name
```

The address specified for the segment in a locating control is not compatible with the segment's alignment. The segment is placed at the specified address, violating its alignment.

```
WARNING 4: DATA SPACE MEMORY OVERLAP
FROM:       byte.bit address
TO:         byte.bit address
```

The data space in the given range is occupied by two or more segments.

```
WARNING S: CODE SPACE MEMORY OVERLAP
FROM:       byte address
TO:         byte address
```

The code space in a given range is occupied by two or more segments.

```
WARNING 6: XDATA SPACE MEMORY OVERLAP
FROM:       byte address
TO:         byte eddress
```

The xdata space in the given range is occupied by two or more segments.

```
WARNING 7: MODULE NAME NOT UNIQUE
MODULE:      file-name(module-name)
```

The specified name was used as the module name for more than one module. The specified module is not processed.

```
WARNING 8: MODULE NAME EXPLICITLY REQUESTED FROM
           ANOTHER FILE
MODULE:      file-name(module-name)
```

The specified module was requested, explicitly, to be processed from another file that has not yet been processed. The specified module is not processed.

```
WARNING 9: EMPTY ABSOLUTE SEGMENT
MODULE:      file-name(module-name)
```

The specified module contains an empty absolute segment. This segment is not allocated. The base address of this segment may be overlapped without any additional message.

Errors

```
ERROR 101: SEGMENT COMBINATION ERROR
SEGMENT:     segment-name
MODULE:      file-name(module-name)
```

The attributes of the specified partial segment, in the specified module, contradict those of previous (unspecified) occurrences of partial segments with the same name. The segment is ignored.

```
ERROR 102: EXTERNALS ATTRIBUTE MISMATCH
SYMBOL:      external-name
MODULE:      file-name(module-name)
```

The attributes of the specified external symbol, in the specified module, contradict those of previous (unspecified) occurrences of public symbol with the same name. The specified symbol is ignored.

```
ERROR 103: EXTERNAL ATTRIBUTES DO NOT MATCH PUBLIC
SYMBOL:      symbol-name
MODULE:      file-name(module-name)
```

The attributes of the specified external (public) symbol, in the specified module, contradict those of previous (unspecified) occurrences of public (external) symbol with the same name. The specified symbol is ignored.

```
ERROR 104: MULTIPLE PUBLIC DEFINITIONS
SYMBOL:      symbol-name
MODULE:      file-name(module-name)
```

The specified public symbol, in the specified module, has already been defined in a previously (unspecified) processed module. The specified symbol is ignored.

```
ERROR 105: PUBLIC REFERS TO IGNORED SEGMENT
SYMBOL:      public-name
SEGMENT:     segment-name
```

The specified public symbol is defined referencing the specified ignored segment. The specified public symbol is ignored.

```
ERROR 106: SEGMENT OVERFLOW
SEGMENT:     segment-name
```

The specified segment, after combination, is larger than the maximum segment size allowed for the segment according to its type or to the given locating control. The specified segment is ignored.

```
ERROR 107: ADDRESS SPACE OVERFLOW
SPACE:       space-name
SEGMENT:     segment name
```

RL51 was unable to allocate the specified relocatable segment, according to the segment relocation type, in the specified address space. The specified segment is ignored.

```
ERROR 108: SEGMENT IN LOCATING CONTROL CANNOT BE
           ALLOCATED
SEGMENT:     segment name
```

RL51 was unable to allocate the specified relocatable segment that appears in the locating control, according to the requirements imposed by the locating control and according to the segment relocation type. The specified segment is ignored.

```
ERROR 109: EMPTY RELOCATABLE SEGMENT
SEGMENT:     segment-name
```

The specified segment, after combination has zero size. The specified segment is ignored.

```
ERROR 110: CANNOT FIND SEGMENT
SEGMENT:     segment-name
```

The specified segment name occurred in the command tail but is not the name of any segment defined within the input files. The specified segment is ignored.

```
ERROR 111: SPECIFIED BIT ADDRESS NOT ON BYTE BOUNDARY
SEGMENT:     segment-name
```

The specified segment was requested in a BIT locating control. The segment is not a BIT segment, and the requested address is not on byte boundary. The specified segment is ignored.

```
ERROR 112: SEGMENT TYPE NOT LEGAL FOR COMMAND
SEGMENT:     segment-name
```

The specified segment is not one of the types that are legal for the locating control for which it is specified. The specified segment is ignored.

```
ERROR 113: RESERVED.
```

ERROR 121: IMPROPER FIXUP
MODULE: *file-name(module-name)*
SEGMENT: *segment-name*
OFFSET: *pseg-offset*

An error occurred in the evaluation of a fixup. An example of this error is when the value of the fixup expression does not meet the requirements of the type of the referenced location. A fixup is an address that cannot be determined at compile/assembly-time. It is marked as relocatable, and at RL-51 time is assigned an address. A fixup is the address assigned to that relocatable symbol.

FATAL ERROR 208: INVALID FILE NAME
partial command

The file-name specified in the command is not a valid file name. The command is repeated up to and including the point or error.

FATAL ERROR 209: FILE USED IN CONFLICTING CONTEXTS
FILE: *file-name*

The specified file is used in more than one context, for example, using the same file for both input and output. (This may be caused by specifying for the first input file a file that has no extension, and not specifying an output file.)

FATAL ERROR 210: I/O ERROR, INPUT FILE;
UDI ERROR: EXCEPTION <num>:<ext>
FILE: *file-name*

An I/O error was detected in accessing an input file. The text of the message includes a description of the specific I/O error that occurred. See the user's guide for your operating system for a list of possible I/O errors.

FATAL ERROR 211: I/O ERROR, OUTPUT FILE; ERROR*
FILE: *file-name*

An I/O error was detected in accessing the output file. The text of the message includes a description of the specific I/O error that occurred. See the user's guide for your operating system for a list of possible I/O errors.

FATAL ERROR 212: I/O ERROR, LISTING FILE; ERROR*
FILE: *file-name*

An I/O error was detected in accessing the listing file. The text of the message includes a description of the specific I/O error that occurred. See the user's guide for your operating system for a list of possible I/O errors.

FATAL ERROR 213: I/O ERROR, TEMPORARY FILE; ERROR*
FILE: *file-name*

An I/O error was detected in accessing a temporary file. The text of the message includes a description of the specific I/O error that occurred. See the user's guide for your operating system for a list of possible I/O errors.

FATAL ERROR 214: INPUT PHASE ERROR
MODULE: *file-name(module-name)*

This error occurs when RL51 encounters different data during pass two than it read during pass one.

FATAL ERROR 215: CHECK SUM ERROR
MODULE: *file-name(module-name)*

A bad check sum was detected in the input module. This indicates a bad input module or a read error.

FATAL ERROR 216: INSUFFICIENT MEMORY

The memory available for execution of RL51 has been used up. This is usually caused by too many external /public symbols or segments in the input files or by too many errors.

FATAL ERROR 217: NO MODULE TO BE PROCESSED

After scanning all the input files, no module was selected to be processed. This is usually caused by an empty input file(s) or incorrect module names in the input list.

FATAL ERROR 218: NOT AN OBJECT FILE
FILE: *file-name*

The file named in the message, judging by its first byte of data, is not a valid object file.

FATAL ERROR 219: NOT AN 8051 OBJECT FILE
FILE: *file-name*

The translator-ID field in the module header record indicates that the specified module is not an 8051 object module.

FATAL ERROR 220: INVALID INPUT MODULE
MODULE: *file-name(module-name)*

The specified input module was found to be invalid. Possible causes are incorrect record order, incorrect record type, illegal field, illegal relation between fields, or a missing required record. This error could be the result of a translator record.

FATAL ERROR 221: MODULE SPECIFIED MORE THAN ONCE
partial command

The input list in the invocation line contains the same module name more than once. The command is repeated up to and including the point of error.

FATAL ERROR 222: SEGMENT SPECIFIED MORE THAN ONCE
partial command

The locating controls in the invocation line contain the same segment name more than once. The command is repeated up to and including the point of error.

FATAL ERROR 224: DUPLICATE KEYWORD
partial command

The same keyword appears in the command more than once. The command is repeated up to and including the point of error.

```
FATAL ERROR 225: SEGMENT ADDRESSES ARE NOT IN
                 ASCENDING ORDER
```
partial command

The addresses of the segments within one locating control are not in ascending order. The command is repeated up to and including the point of error.

```
FATAL ERROR 226: SEGMENT ADDRESS INVALID FOR CONTROL
```
partial command

The address requested for a segment is not valid for the given locating control. The command is repeated up to and including the point of error.

```
FATAL ERROR 227: PAGEWIDTH PARAMETER OUT OF RANGE
```
partial command

The PAGEWIDTH parameter given is out of the acceptable range.

```
FATAL ERROR 228: RAMSIZE PARAMETER OUT OF RANGE
```
partial command

The RAMSIZE parameter given is out of acceptable range.

```
FATAL ERROR 229: I/O ERROR, OVERLAY FILE; ERROR#
FILE:           file-name
```

An I/O error was detected in accessing an overlay file. The text of the message includes a description of the specific I/O error that occurred. See the user's guide for your operating system for a list of possible I/O errors. (This error occurs only if IXREF was requested. Its occurrence does not invalidate the output object file.)

```
FATAL ERROR 230: INCOMPATIBLE OVERLAY VERSION
FILE:           file-name
```

The overlay file, although loaded successfully, has a version number that is not the one expected by RL51. The possible cause is that the RL51 program and the loaded overlay are not of the same version. (This error occurs only if IXREF or OVERLAY was requested. If only IXREF was requested, the output object file is valid.)

```
FATAL ERROR 231: TOO MANY IXREF ENTRIES
```

The number of IXREF entries (entry is a pair consisting of modules and symbol reference) is too large to be processed. The IXREF listing step is not performed. The NOLIBRARIES and NOGENERATED controls may be used in order to decrease this number and overcome the error. (This error occurs only if IXREF was requested. Its occurrence does not invalidate the output object file.)

```
FATAL ERROR 232: OVERLAY CONTROL CONFLICTS-XREF
SELECTOR ITEMS
```

The overlay control should not appear with the IXREF selector items NOLIBRAR-IES or NOGENERATED.

```
FATAL ERROR 233: ILLEGAL USE OF * IN OVERLAY
CONTROL
```

The use of " > " or "] " with the OVERLAY control is illegal.

```
FATAL ERROR 240: INTERNAL PROCESS ERROR
```

RL51 has detected that it has made a processing error. This error indicates a bug within RL51.

Table C-1. LIB51 Commands

Command	Abbreviation	Description
ADD { file[(module[,....])] } [....] TO library_file	A	Adds modules to a library
CREATE library file	C	Creates a library file
DELETE library_file(module[,....])	D	Deletes modules from a library
EXIT	E	Terminates session with LIB51
EXTRACT { file[(module[,....])] } [....] TO file	X	Extracts modules from libraries
HELP	H	Displays syntax of LIB96 commands
LIST { file[(module[,....])] } [....] [TO file] [PUBLICS]	L [P]	Lists modules contained in libraries, and optionally lists all publics
REPLACE { file[(module[,....])] } [....] IN library_file	R	Replaces modules in a library

INSUFFICIENT MEMORY

LIB51 cannot execute the command because it requires more memory than the amount of memory available in the system.

INVALID SYNTAX

The command was not entered properly. Reenter it using the correct syntax.

UNRECOGNIZED COMMAND

An illegal or misspelled command was entered. The only commands are ADD, CREATE, DELETE, EXIT, EXTRACT, HELP, LIST, REPLACE, and their respective abbreviations.

INVALID MODULE NAME

The specified module name contains an invalid character or starts with a digit.

MODULE NAME TOO LONG

The specified module name exceeds 40 characters.

RIGHT PARENTHESIS EXPECTED

A ")" is missing in the command.

pathname, CHECKSUM ERROR

The specified file has an error in one of its checksum fields. This is usually the result of an I/O error.

pathname, ILLEGAL RECORD FORMAT

This error is usually caused by an I/O error or a translation error.

pathname, BAD RECORD SEQUENCE

This error is usually caused by an I/O error or a translation error.

pathname, DUPLICATE SYMBOL IN INPUT

You have attempted to ADD or REPLACE a module that contains a public symbol that is already within the library.

pathname, ATTEMPT TO ADD DUPLICATE MODULE

The specified module name already appears within the library.

pathname, FILE ALREADY EXISTS

The specified file in the CREATE command already exists. Choose another name for the library.

pathname, NOT LIBRARY

The specified file is not a library.

The TO filename is omitted in the ADD command.

UNRECOGNIZED COMMAND

An illegal or misspelled command was entered. The only legal commands are ADD, CREATE, DELETE, LIST, and EXIT.

File or Module Errors

The following errors indicate that there is some problem with the file or module specified. There is no partial copy of the command given with these error messages.

FILE ALREADY EXISTS

The file specified in the CREATE command already exists. Choose a new name for the library.

filename, DUPLICATE SYMBOL IN INPUT

You have attempted to add a file that contains a PUBLIC symbol already within the library.

filename, NOT LIBRARY

The specified file is not a library.

filename(*modname*): NOT FOUND

You have attempted to delete a module that does not exist. Check for misspelling of the filename or module name.

modname—ATTEMPT TO ADD DUPLICATE MODULE

The specified module name already appears within the library.

symbol—ALREADY IN LIBRARY

You have attempted to add a module that contains a PUBLIC symbol that is already in the library.

filename, CHECKSUM ERROR

filename, OBJECT RECORD TOO SHORT

filename, ILLEGAL RECORD FORMAT

LIB51 cannot process the specified file because it is not a legal object file. Possible cause is a file damage or translator error.

Hexadecimal-Decimal Conversion Table

E

Table E-1 is for hexadecimal to decimal and decimal to hexadecimal conversion. To find the decimal equivalent of a hexadecimal number, locate the hexadecimal number in the correct position and note the decimal equivalent. Add the decimal numbers.

To find the hexadecimal equivalent of a decimal number, locate the next lower decimal number in the table and note the hexadecimal number and its position. Subtract the decimal number shown in the table from the starting number. Find the difference in the table. Continue this process until there is no difference.

Table E-1. Hexadecimal-Decimal Conversion Table

Most Significant Byte				Least Significant Byte			
Digit 4		Digit 3		Digit 2		Digit 1	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4 096	1	256	1	16	1	1
2	8 192	2	512	2	32	2	2
3	12 288	3	768	3	48	3	3
4	16 384	4	1 024	4	64	4	4
5	20 480	5	1 280	5	80	5	5
6	24 576	6	1 536	6	96	6	6
7	28 672	7	1 792	7	112	7	7
8	32 768	8	2 048	8	128	8	8
9	36 864	9	2 304	9	144	9	9
A	40 960	A	2 560	A	160	A	10
B	45 056	B	2 816	B	176	B	11
C	49 152	C	3 072	C	192	C	12
D	53 248	D	3 328	D	208	D	13
E	57 344	E	3 548	E	224	E	14
F	61 440	F	3 840	F	240	F	15

REQUEST FOR READER'S COMMENTS

Please use this form to help us evaluate the effectiveness of this manual and improve the quality of future documents.
(Please limit your comments to documentation.)

To order publications, contact the Intel Literature Department (see page ii of this manual).

Indicate the type of user/reader you most nearly represent:

☐ Software engineering manager
☐ Systems engineer
☐ Application programmer

☐ Student programmer
☐ Other (please specify) _____

Indicate the application for which you use this product: _____
_____
_____

How do you use this manual?

☐ As a reference manual
☐ As an introduction to the subject

☐ Other (please specify) _____

Fill in the squares below with e rating of 5 (excellent) to 1 (poor); use the lines provided for edditional comments.

☐ **Accuracy.** Cite any instances of inaccuracies. _____
_____

☐ **Organization.** Could you find the information you needed? _____
_____

☐ **Thoroughness.** Are product features fully explained? _____
_____

☐ **Readability.** Is the writing style clear? _____
_____

☐ **Examples.** Are there enough examples? _____
_____

☐ **Level.** Is the manual too easy or too difficult? _____
_____

☐ **Overall.** How would you improve this manual? _____
_____
_____
_____

List specific points that need clarification:

Page   Problem

_____   _____
_____   _____
_____   _____

Your Name _____ Job Title _____
Company Name _____
Street Address _____
City _____ State _____ Zip _____

☐ Check here if you want e written reply.